
**BRNO UNIVERSITY OF TECHNOLOGY**
**VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ**

**FACULTY OF INFORMATION TECHNOLOGY**
**FAKULTA INFORMAČNÍCH TECHNOLOGIÍ**

**DEPARTMENT OF INFORMATION SYSTEMS**
**ÚSTAV INFORMAČNÍCH SYSTÉMŮ**


# AUTOMATIC WEB PAGE RECONSTRUCTION

**AUTOMATIZOVANÁ REKONSTRUKCE WEBOVÝCH STRÁNEK**

**MASTER'S THESIS**
**DIPLOMOVÁ PRÁCE**


**AUTHOR** **VILIAM SEREČUN**
**AUTOR PRÁCE**

**SUPERVISOR** **Ing. VLADIMÍR VESELÝ, Ph.D.**
**VEDOUCÍ PRÁCE**



**BRNO 2018**

*Zadání diplomové práce/21138/2017/xserec00*

**Vysoké učení technické v Brně - Fakulta informačních technologií**

Ústav informačních systémů Akademický rok 2017/2018

# Zadání diplomové práce

Řešitel: **Serečun Viliam, Bc.**
Obor: Bezpečnost informačních technologií
Téma: **Automatizovaná rekonstrukce webových stránek**
**Automatic Webpage Reconstruction**
Kategorie: Počítačové sítě

Pokyny:

1. Seznamte se s problematikou analýzy, rekonstrukce a zejména exportu webového provozu - protokol HTTP, HTTPS, internetové proxy a exportní formáty (zejména MHT a MAFF).
2. Navrhněte nástroj, který bude pravidelně ukládat vybrané webové stránky do MAFF souboru. Promyslete i možnosti parsování webového obsahu a následné uložení metadat.
3. Podle doporučení vedoucího implementujte nástroj a webový front-end k němu, ve kterém si bude možné plánovat úlohy a parametry rekonstrukce.
4. Otestujte a analyzujte vytvořené řešení na reprezentativním vzorku stránek, a to se zaměřením na kryptoměnové weby. Diskutujte jeho škálovatelnost do budoucna.

Literatura:


- R. Fielding, "RFC 2616 - Hypertext Transfer Protocol -- HTTP/1.1", IETF, 1999.
- H. Lie, "The text/css Media Type", IETF, 1998.
- W3C, "Document Object Model (DOM)", online: http://www.w3.org/DOM/, 2011.
- Mozdev Community Organization Inc., "The MAFF Specification", online: http://maf.mozdev.org/maff-specification.html, 2014.
- Bitcoin Project, *Developer Guide - Bitcoin*, dostupné na: https://bitcoin.org/en/developer-guide.


Podrobné závazné pokyny pro vypracování diplomové práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva diplomové práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap, které byly vyřešeny v rámci dřívějších projektů (30 až 40% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Veselý Vladimír, Ing., Ph.D.**, UIFS FIT VUT
Datum zadání: 1. listopadu 2017
Datum odevzdání: 23. května 2018

L.S.

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav informačních systémů
612 66 Brno, Božetěchova 2

doc. Dr. Ing. Dušan Kolář
*vedoucí ústavu*

# Abstract

Many legal institutions require a burden of proof regarding web content. This thesis deals with a problem connected to web reconstruction and archiving. The primary goal is to provide an open source solution, which will satisfy legal institutions with their requirements. This work presents two main products. The first is a framework, which is a fundamental building block for developing web scraping and web archiving applications. The second product is a web application prototype. This prototype shows the framework utilization. The application output is MAFF archive file which comprises a reconstructed web page, web page screenshot, and meta information table. This table shows information about collected data, server information such as IP addresses and ports of a device where is the original web page located, and time stamp.



# Abstrakt

Mnoho právnych inštitúcií vyžaduje dôkazné bremeno týkajúce sa webového obsahu. Táto diplomová práca sa zaoberá problémom spojeným s automatizáciou webovej rekonštrukcie a webovou archiváciou. Hlavným cieľom je poskytnúť riešenie s s otvoreným zdrojovým kódom , ktoré uspokojí právne inštitúcie s ich požiadavkami. Táto práca predstavuje dva hlavné produkty. Prvý je rámcový program, ktorý je základným stavebným kameňom pre vývoj aplikácií na extrakciu a archiváciu webových stránok. Druhým produktom je prototyp webovej aplikácie. Tento prototyp ukazuje využitie rámcového programu pri riešení požiadavok týchto inštitúcií. Výstupom aplikácie je archív formátu MAFF, ktorý obsahuje zrekonštruovanú webovú stránku, snímku obrazovky webovej stránky a tabuľku meta-informácií. Táto tabuľka zobrazuje informácie o zhromaždených údajoch, informáciách o serveroch, ako sú napríklad IP adresy a porty zariadenia, na ktorom sa nachádzala pôvodná webová stránka, a časové razítko.



# Keywords

Web archiving, Mozilla Archive Format, Web scraping, Web indexing, Lemmiwinks, Multi-Functional Index Scraping Tool

# Klíčová slova

Webová archivácia, Mozilla Archive Format, Extrakcia webu, Webová indexácia, Lemmiwinks, Multi-Funkcionálny nástroj na indexáciu a archiváciu webového obsahu


# Reference

SEREČUN, Viliam. *Automatic Web Page Reconstruction.* Brno, 2018. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Vladimír Veselý, Ph.D.

# Rozšířený abstrakt

S novou formou trestnej činnosti, ktorá sa vyvíja na internete, ako je trh s drogami alebo obchodovanie s ľuďmi, právne inštitúcie požadujú, aby sa zhromažďovali a archivovali uverejnené informácie zo strany počítačových kriminálnikov. Neexistuje však riešenie s otvoreným zdrojom kódom, ktoré by ponúkalo túto funkčnosť.

V tejto práci predstavujem prístupy k rekonštrukcii webu. Existujú dva hlavné problémy. Prvým je výber webového archívu. Neexistuje štandardný archívny formát a webové prehliadače plne podporujú len niekoľko z nich. Druhým problémom je štruktúra HTML dokumentov a zložitosť súvisiaca s modernými webovými stránkami, najmä s ich dynamickým obsahom.

Väčšina súčasných nástrojov a riešení poskytuje čiastočnú funkcionalitu súvisiacu s rekonštrukciou webu a automatizovanou extrakciou dát z webu. Komerčné riešenia sú väčšinou zamerané na extrakciu webového obsahu, ako je Dexi.io [1] alebo Scrapinghub[2], a otvorené riešenie [3] poskytuje podobný prístup. Existujú aj rozšírenia prehliadača pre archiváciu webu [17, 8], ale nie je možné ich použiť na automatizáciu.

Žiadne z týchto riešení však neposkytuje komplexnú platformu. Preto som vytvoril som rámcovú a webovú aplikáciu, kde som sa pokúsil pripojiť archiváciu, extrakciu a indexovanie obsahu webových stránok. Rámec Lemmiwinks poskytuje súbor nástrojov na archiváciu obsahu a získavanie informácií z neho. Pomocou tohto rámcu som vyvinul webovú aplikáciu, ktorá má schopnosť naplánovať extrakciu a archiváciu úloh. Táto aplikácia slúži aj ako vyhľadávacia platforma, v ktorej môže používateľ hľadať uložené informácie a dokonca vizualizovať snímku webových stránok.

Moje súčasné riešenie poskytuje veľa možností na vývoj automatizačného nástroja na archiváciu webového obsahu a získavanie informácií z neho. Rámec podporuje prístup k webovým stránkam s predpracovaním webového obsahu a bez neho. Aktuálne podporovaný archívny formát je Mozilla Archív Format. Webová aplikácia MultiFIST (Multifunkčný indexový extrakčný nástoj) je ukážkou možného budúceho produktu. Cieľom prístupu k archivovaným webovým stránkam bolo vytvoriť pocit, že správanie archivovanej webovej stránky je rovnaké ako na internete.

Rámec však má určité obmedzenia, na webové stránky chránené programom Captcha alebo inými pokročilými nástrojmi. Tieto obmedzenia spôsobujú problémy, pri pokuse o rekonštrukciu webového obsahu. Existujú aj obmedzenia pri získavaní údajov pomocou regulárneho výrazu. Chyba je založená na sémantike údajov a nemožnosti rozlíšiť význam obsahu pomocou regulárneho výrazu.

Výhodou môjho riešenia je flexibilita pri vývoji aplikácií. Vývojár môže použiť rámec na optimalizáciu požiadaviek. Rámec tiež poskytuje sofistikovanú platformu pre archiváciu webových stránok a extrakciu webov, ktoré iné riešenia neposkytujú. Táto práca bola vykonaná ako súčasť riešenia projektu TARZAN, na ktorom sa zúčastňujem.

# Automatic Web Page Reconstruction

## Declaration

Hereby I declare that this diploma's thesis was prepared as an original author's work under the supervision of Mr. Ing. Valdimír Veselý Phd. All the relevant information sources, which were used during preparation of this thesis, are properly cited and included in the list of references.

.....
Viliam Serečun
May 23, 2018


## Acknowledgements

I would like to thank to my supervisor Mr. Ing. Vladimír Veselý Phd. for his help and suport.

# Contents

# List of Figures

# Chapter 1

# Introduction

With a new form of crime which is evolving on the Internet such as a drug market or people trafficking, legal institutions demand to collect, and archive published information by cybercriminals. However, there is no open source solution to offer this functionality.

In this thesis, I introduce web reconstruction approaches. There are two main related problems. The first is a selection of web archive format. There is not a standard archive format, and web browsers fully support only a few of them. The second problem is an HTML document structure and complexity related to modern web pages, especially with dynamic content.

Most of the current tools and solutions give partial functionality related to web reconstruction and web scraping automation. Commercial solutions are mostly focused on web scraping such as Dexi.io [1] or Scrapinghub [2], and open source solution [3] provides similar approach. There are also browser extensions [17, 8] for web archiving, but it is not possible to use them for automation.

However, any of these solutions provide a comprehensive platform. I developed a framework and web application, where I tried to connect archiving, scraping and indexing of web page content. The Lemmiwinks framework provides a set of tools to archive content and extract information from it. Upon this framework, I developed a web application having an ability to schedule scraping and archive tasks. This application also serves as a search platform, where a user can look for stored information and even visualize the web page snapshot.

My current solution provides many possibilities to develop automation tool to archive web content and extract information from it. The framework supports accessing websites with and without JavaScript rendering of web content. The currently supported archive format is Mozilla Archive Format. The web application MultiFIST (Multi-Functional Index Scraping Tool) is a demonstration of the possible future product. The goal of accessing archived website was to create feeling that behavior of an accessed archived web page is the same as the original one stored on the Internet.

## 1.1 Chapters overview

Chapter 2 focuses on web reconstruction. This chapter describes web reconstruction modes, a structure of HTML document and elements. It also shows some of these elements and demonstrates their attributes or URL path resolve.

The next chapter focuses on implementation details of scrapping and archiving framework. This chapter should be used as technical documentation for developers.

Chapter 4 describes console and web applications and their use. It also describes the design and architecture of a particular part of their implementation.

Following chapter 5 shows, how the application was tested. It also demonstrates potential usage or even parts which, will need some improvements.

The last chapter evaluate result of this thesis and discuss future fork.

# Chapter 2

# Web Reconstruction

There are two approaches related to web page reconstruction during its accessing:

- with JavaScript execution, when the content is reconstructed with dynamic elements already rendered. A drawback of this solution is that it requires more CPU power and it is slow, but it provides better results on dynamic web content.
- without JavaScript execution. The web page is not rendered before archiving, which can cause a problem with dynamic content rendering. This approach is extremely fast because it requires just HTTP GET requests when the received content is stored directly into an archive.

The following sections of this chapter discuss archive format used lately in the implementation part of this thesis. This chapter also describes some difficulties found during the reconstruction of the web content such as HTML document structure recursion or some HTML element characteristics. The last section shows how to resolve URL addresses included in some element attributes.

## 2.1 Mozilla Archive Format

Mozilla Archive Format (MAFF) was designed to store multiple web pages into a single archive. The archive is an ordinary ZIP file. The design of this format even allows storing dynamic content like videos or audio files. It also allows storing meta-data of saved resources [6].

Comparing to other formats and mostly to MHTML archiving format, MAFF has several advantages. It is easy to implement because the main idea is to download the web document tree structure, while in the MHTML format all data (even binary) are stored in one file. This structure is compressed, and if a browser does not support this format, it is still possible to access reconstructed web content by uncompressing the ZIP file and accessing the root index file. This format also supports multiple tab archiving, which allows to archive more related information into one archive. Probably the most significant advantage of MAFF is simplicity, and a possibility to access archive content without specialized software [6].

### 2.1.1 Technical characteristics

MAFF specification defines four conformance levels. Elementary, basic, normal and extended levels determine different requirements for reading and writing the implementation

of the standard. The main differences among these levels are specifications for storing and accessing meta-data in archives [6].

Figure 2.1 shows the directory structure of MAFF archive satisfying basic level. MAFF archive contains a root directory. The root directory has 1 to N sub-directories. Each sub-directory represents one tab. Inside the sub-directory are placed the index file, optionally *index.rdf*, and *index_files* directory. The file extension of the index file depends on the content type. If the index file consists of external resources, *index_files* directory will store these resources. The*index.rdf* file stores meta-information collected from a web document [6].

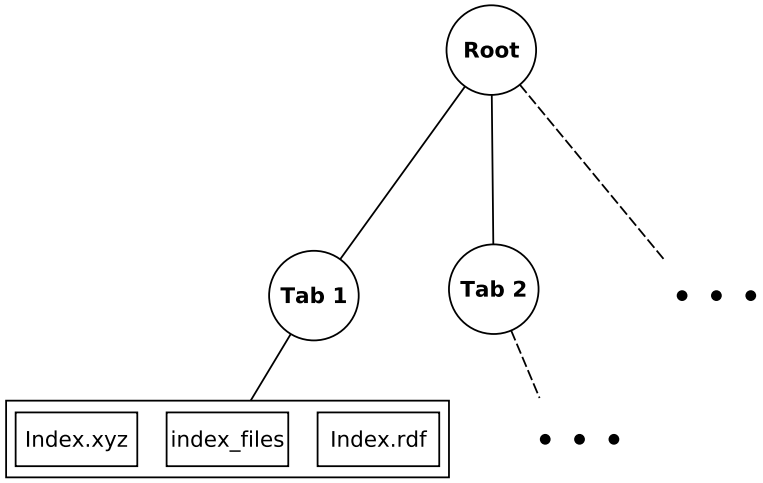



Figure 2.1: MAFF directory structure

An example of the content of a *index.rdf* file is shown in Figure 2.2. The content has XML-based syntax. The RDF file usually includes the following information [7]:

- The file name of the main document.
- The original URL of the page.
- Date and time of the save operation.
- The title of the page, if present.
- The character set to use when parsing files that are part of the page.

## 2.2 Web Page Structure

Generally, it is possible to divide the HTML document into three parts. The first part represents general HTML elements. These elements create and describe document content. They contain attributes, and their values can include external references to binary data or even to other HTML documents.

Cascading Style Sheets (CSS) specifies web page presentation. The CSS code can be present in HTML element attribute style. This mode is declarative. Another way how to

```
<RDF:RDF>
    <RDF:Description RDF:about="urn:root">
        <MAF:originalurl RDF:resource="https://example.com/"/>
        <MAF:title RDF:resource="Example"/>
        <MAF:archivetime RDF:resource="Thu 05 Oct 2017 23:55:57 +0200"/>
        <MAF:indexfilename RDF:resource="index.html"/>
        <MAF:charset RDF:resource="utf-8"/>
    </RDF:Description>
</RDF:RDF>
```

Figure 2.2: RDF file example

include CSS code inside HTML document is by specifying a tag style or link. The link element refers to the external CSS file. CSS code can include other CSS files or other references to binary data [19, 5].

The last part of an HTML document is dynamic content. JavaScript is responsible for the dynamic content. JavaScript can be present in the HTML document inside script tags, as a reference to an external file, or inside the HTML elements as an event. These scripts can also dynamically create new elements depending on user behavior.

Figure 2.3 presents possible parts of web page document, which can cause recursion. During the website content reconstruction, it is necessary to consider these aspects. Ignoring recursive resources can produce insufficient results.

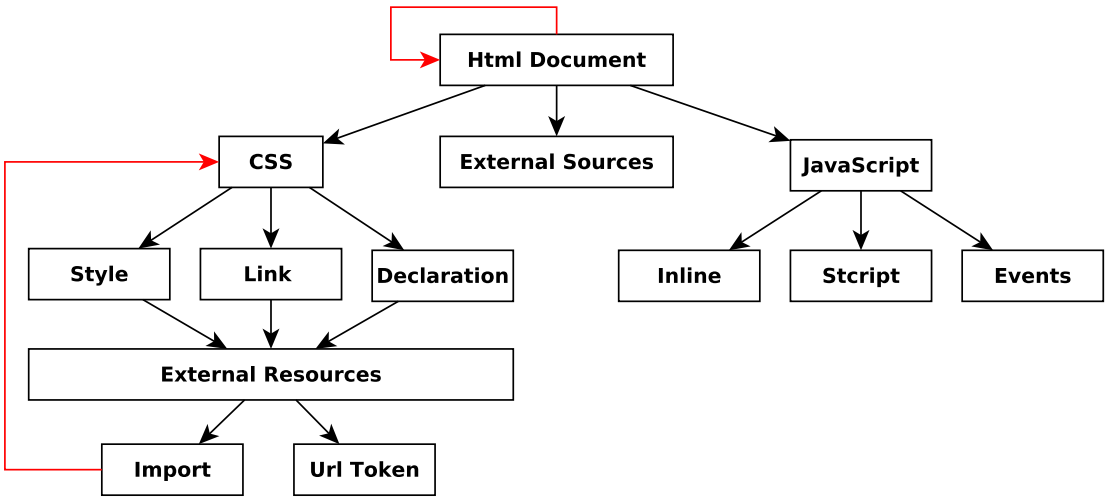



Figure 2.3: Web Page document structure. Red arrows represent recursion where the resources can refer to other resources.

### 2.2.1 HTML Elements

Web page reconstruction requires processing of several HTML elements. These elements are listed in Table 2.1. The elements are closely related to the attributes. All listed attributes,

if they are present in the element, include reference to external data. These data are either binary files or other documents, which need to be processed recursively [19].

| Element | Attributes |
|---|---|
| img | src; data-src |
| video | src; poster |
| embed | src |
| source | src |
| audio | src |
| input | src |
| object | data; codebase |
| track | src |
| link | href; rel=stylesheet |
| script | src |
| script | - |
| style | - |
| frame | src |
| iframe | src |
| all | style |

Table 2.1: HTML elements and their attributes

For better understanding Figure 2.4 shows some example of HTML elements containing a reference to an external entity. The attribute reference value on the second line of the figure shows a particular type of reference. This reference contains binary data encoded in the Base64 scheme [18].

```
1 <img src="/wp-content/uploads/flamingo.jpg">
2 <img src="...==">
```

Figure 2.4: HTML elements referencing external resources.

JavaScript or CSS code can also be present inside the element body. The CSS code processing in the element body is the same as for CSS declaration, or an external CSS file. The difference is with JavaScript processing, which depends on the web reconstruction mode. If the web content was pre-rendered, then it is necessary to remove all scripts to avoid double-rendering of some elements. Otherwise, the JavaScript is typically reconstructed. The same approach is necessary when the element includes the event attribute. A different form of JavaScript representation inside a HTML document is shown in Figure 2.5. All event attributes are listed in Appendix A.

### 2.2.2 CSS Elements

Two types of CSS elements are necessary to distinguish during processing of CSS. The first ones are URL tokens, which refer to external data (mostly binary). The second type of symbols are at-rules or import rules which contain a reference to another CSS file. These tokens can cause recursion of CSS files when one CSS file refers to another. Figure 2.6 shows both examples of CSS tokens.

(a) HTML element referencing a JavaScript file

```
<script type="text/javascript" src="veverka.js"></script>
```

(b) Inline JavaScript

```
<script type="text/javascript">
   $(function(){
       document.getElementById('click_me').onclick =
           function () { alert('VEVERKA'); };
   });
</script>
```

(c) HTML element with JavaScript event

```
<button onclick="displayVeverka()">VEVERKA</button>
```

Figure 2.5: References to JavaScript code inside a HTML document.

(a) At-Rules with URL and string tokens.

```
1 @import "veverka.css";
2 @import url("/veverka.css");
```

(b) URL Token

```
1 background: url(veverka.gif);
```

Figure 2.6: Example of URL and string tokens in CSS code.

## 2.3 URL and Path resolving

URL addresses can reference external resources. The references can have various forms. The element's attributes value can reference data using a full URL address with or without URI scheme, or they can contain a relative path. The relative path can have two representations depending on the starting character. The example in Figure 2.7 shows clearly how the relative path can look like and how the URL address is resolved. The paths and incomplete URL addresses are resolved using the base URL address, which can be specified by the HTML element, or it can be the actual URL address of the HTML document [18].

An Internationalized Resource Identifier (IRI) is an URL representation using national characters of some country. This representation can cause a problem when accessing these addresses, as not all HTTP clients can automatically decode them. The solution is to decode national characters into a hexadecimal representation that the HTTP client can process. Figure 2.8 shows an example of IRI address and its hexadecimal representation [15].

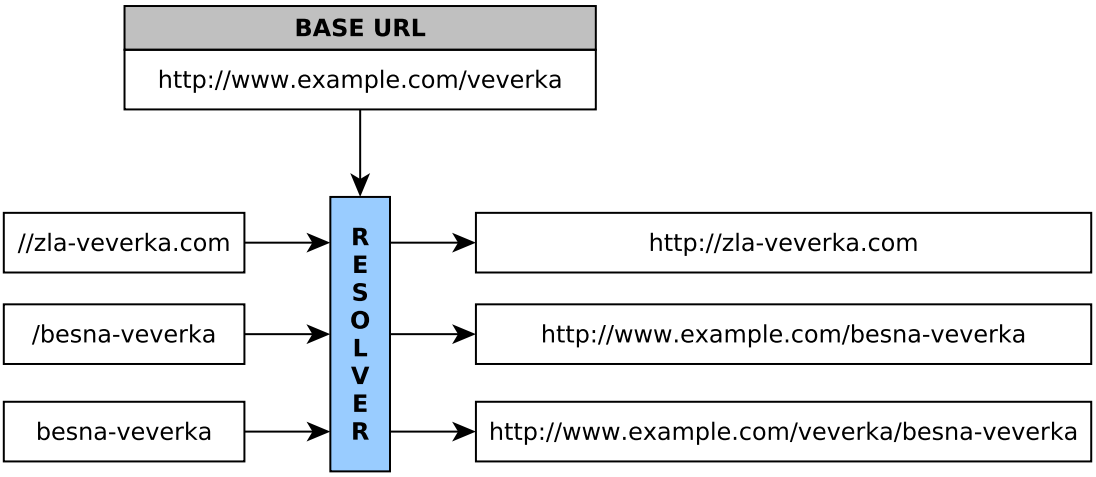



Figure 2.7: URL resolving example

https://en.wiktionary.org/wiki/Ῥόδος

==

https://en.wiktionary.org/wiki/%E1%BF%AC%CF%8C%CE%B4%CE%BF%CF%82

Figure 2.8: IRI address and its hexadecimal representation.

# Chapter 3

# Framework

The Lemmiwinks is a framework providing scraping and archiving functionalities. The advantage of this framework is the possibility to file the dynamic content of websites. The archiving format is the MAFF, which I chose for the opportunity to archive several tabs in one document. However, the current implementation of the framework does not support CAPTCHA resolving, and visiting private content, accessible by signing in to some platform. These missing functionality parts can cause insufficient reconstruction results.

The following sections of this chapter explain the design and implementation decisions. The implementation part can be used as a reference guide for future project extending.

## 3.1 Design and Architecture

The framework architecture uses mostly dependency injection, factory, singleton and inversion of control design patterns. It is possible to divide the current framework implementation into four parts, where each of them consists of several classes solving a specific problem. For example, the parser package processes CSS and HTML files into small fragments. The modules are listed below, and following subsections describe them in more detail.

- HTTP Library
- Parser
- Archive
- Extractor

The architecture design also allows adding new modules or extend existing ones to satisfy future requirements. This possibility is due to modular and generic design, which defines and uses interfaces for solving particular problems.

The Lemmiwings framework uses Python 3 programming language, requiring version 3.6. Some parts of the framework are paralleled to increase effective use of resources. It uses asyncio standard library, which means, that tasks and coroutines divide the program into small parts handled by task manager. This approach reacts to interruptions made by the operation system, mostly during I/O operations, and provides asynchronous tasks execution. The asynchronous tasks are more efficient than using standard threads, mostly because of the problems related to Global Interpreter Locker. The asyncio library provides a simple way, how to write asynchronous code comparing to other third-party libraries like Tornado or Twisted. The framework also uses various external libraries listed below.

- tinycss2 [24]
- aiofiles [25]
- selenium [4]
- dependency-injector [16]
- validators [26]
- python-magic [14]
- beautifulsoup4 [22]
- lxml [23]
- asyncio-extras [12]
- aiohttp [9]

All dependencies are possible to install from the *requirements.txt* file or *setup.py* file using pip tool [13] for installing and managing Python packages. These files are in the project's root directory.

## 3.2 HTTP library

This module consists of two types of clients: the simple client and the JavaScript asynchronous client. All clients define methods for sending HTTP GET requests and receiving responses from requested servers. The asynchronous JavaScript client differs from the simple client, by being able to render website content, and take a snapshot of the accessed web page. Figure 3.1 shows a class diagram of the whole package.

### Asynchronous client

The asynchronous client consists of abstract class *AsyncClient* and concrete implementation, which is in *AIOClient* class. The abstract class defines several public arguments specifying HTTP client behavior, and two abstract methods. The arguments are:

- *proxy* – specifies IP address of a proxy server.
- *cookies* – represent data sent by the client in the HTTP header.
- *headers* – consist of values sent in the HTTP header.
- *timeout* – specifies a time range, when the client waits for a server response.

The *AIOClient* class inherits from *AsyncClient* class and uses the asyncio standard library with the aiohttp library. These libraries provide the possibility to send thousands of HTTP GET requests asynchronously. The public parameter of this class together with inherited ones is:

- *pool_limit* – represents the maximum open connections per session.

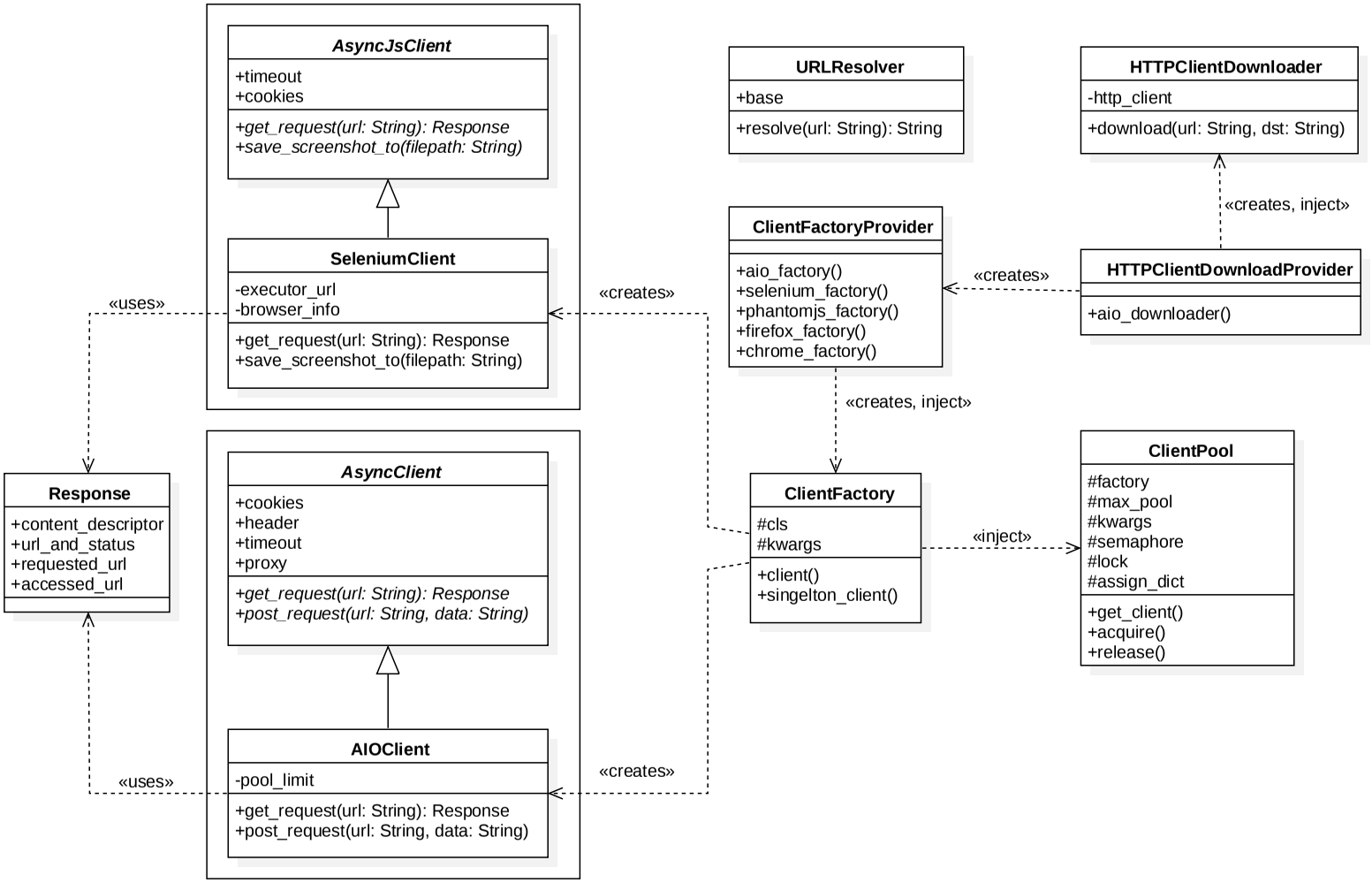



Figure 3.1: Class diagram of HTTP library package.

The class implements these methods:

- *get_request* – an asynchronous method for creating HTTP POST requests. The argument of this method is a URL address represented as a string. This method returns a *Response* object.
- *post_request* – a request definition.

### Asynchronous JavaScript client

Implementation of the asynchronous client, which renders dynamic web page content, can be found in *SeleniumClient* class. This class inherits methods from abstract *AsyncJSClient* class. The abstract class defines two methods and several public arguments for unified usage of other concrete implementations. *AsyncJSClient* class has these arguments:

- *timeout* – a time range, when the client waits for a server response.
- *cookies* – additional data sent in the HTTP header.

The *SeleniumClient* class uses Selenium library. This class is pseudo asynchronous because Selenium library has block functions and it does not support context switching and tasks. The asyncio library, in this case, uses threads. The *SeleniumClient* class must initiate the following arguments to create a client object:

- *executor_url* – represents the URL address of the remote selenium driver.
- *browser_info* – specifies the browser type.

This class also implements inherited abstract methods for sending HTTP GET requests and taking web page snapshots. These methods are:

- *get_request* – sends HTTP GET request. The argument of this method is a URL address represented as a string value. It returns a Response object.
- *save_screenshot_to* – saves the actual web page screenshot into the specified location in png format.

### Response

*Response* class encapsulates a received response from HTTP servers. It stores each response into a temporary file and saves the received HTTP status code. This approach ensures that the framework can handle a significant amount of data (hundreds or thousands of Megabytes) such as images or videos without getting rid of memory. It is also possible to processes data in-memory, which requires a new implementation of *Response* class, keeping all class parameters. The class also provides a unified way to process responses. Response class has four parameters:

- *content_descriptor* – File descriptor of a temporary file. All data obtained from a server are stored there, except when there is a None value, which is either a default value, or indicates that no data was received from a server.
- *url_and_status* – A list of tuples where the first value is a URL address and the second is a HTTP status code. This parameter also stores redirections.

- *requested_url* – the parameter contains the requested URL address.
- *accessed_url* – due redirections while accessing a website, this parameter stores the accessed URL address.

### Providers

Factory classes should instantiate client objects. The *ClientFactory* class provides the object instantiation. The factory must be injected with a class reference pointing to the concrete client class, with the parameters of this class. The class constructor has two parameters:

- *cls* – reference to the client class.
- *kwargs* – dictionary of parameters (parameter name and parameter value) for client object instantiation.

The factory class has two methods. All methods instantiate a client object:

- *client* – creates a new instance.
- *singelton_client* – the first call creates an instance, and other calls of this method return a reference to the same object.

The *ClientFactoryProvider* class is a container which defines specific factories by injecting *ClientFactory* class. This provider instantiates a factory by creating a simple client and several asynchronous JavaScript clients. This class implements five methods where each method returns a specific factory object.

- *aio_factory* – creates an asynchronous client factory.
- *selenium_factory* – creates an asynchronous JavaScript factory where the arguments of this method have to be the browser type and the URL address of the Selenium remote server.
- *phantomjs_factory* – the same as selenium factory with predefined browser type.
- *firefox_factory* – asynchronous JavaScript client factory with Firefox browser type.
- *chrome_factory* – the factory creates chrome instances of asynchronous JavaScript client.

### Pool Manager

Figure 3.2 shows a flowchart describing the Selenium pool functionality. The pool manages instances of Selenium clients. The input can be a request or a Selenium instance. A running task uses the pool to acquire or release a Selenium client instance. If there is not a free instance in the pool, and Selenium pool does not manage a maximal instance limit, a new client instance is created. Otherwise, a task has to wait in a queue, until some instance is released.

Pool manager implements *ClientPool* class. This class has to be injected by client factory, client parameters, and maximum instance limit. Lock and semaphores synchronize the assigning and releasing client object in manager instance. There are three methods:

- *assign* – Reserves client instance for a specific task so no other task can use the instance.
- *release* – Releases the assigned client instance. The parameter of this method is the client object.
- *get_client* – Assigns and releases the client object inside the "with" block.

### Downloader

There are two classes implemented to serve as a downloader. The first class is *HTTPClient-Downloader* and the second is *HTTPClientDownloadProvider* class. Both types of HTTP clients implement the *get_response* method, which returns a *Response* object. Therefore, the *HTTPClientDownloader* consists of a concrete implementation of the download method for both types of clients. The class also has one parameter *http_client*, which represents an HTTP client instance. The download operation has two parameters: the URL address and the file location (where the required entity will be placed).

Because the *HTTPClientDownloader* class has to be injected with a client instance, the *HTTPClientDownloadProvider* does it. Currently, this provider class implements one method which returns an *HttpClientDownloader* object. The method is:

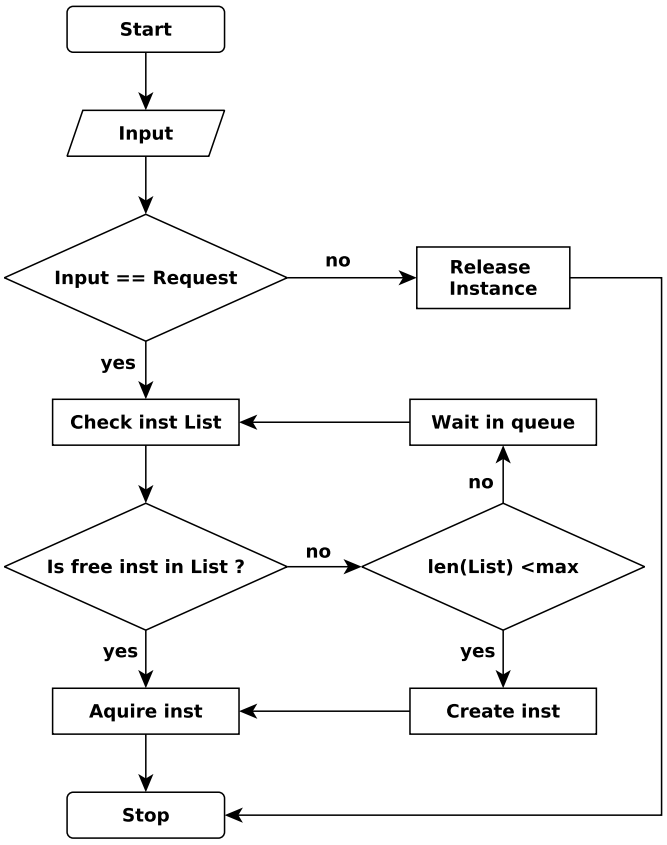



Figure 3.2: Flowchart of Selenium Pool functionality.

- *aio_downloader* – this method instantiates *HTTPClientDownloader* with an *AIOClient* object.

#### Resolver

The *URLReslover* class provides URL resolving towards a specific base URL address. The parameter of this class is the base URL address represented as a string value. This class has one method:

- *resolve* – this method returns a resolved URL address. It has one argument which can be a path or another URL address.

## 3.3 Parser

In the parser module, I implemented two types of parsers. The CSS parser, which searches for import and URL tokens. This parser allows the extraction of external references and their update. HTML parser provides similar functionality. This parser can also extract all textual data from web content to help the Extractor module with data extraction. Figure 3.3 shows the class diagram of the whole python package.

#### HTML parser

It is possible to divide the HTML parser into two parts: the parser and the HTML element representation. The parser currently consists of two classes. The first one is the abstract class, which defines abstract methods as a template for concrete implementation of the HTML parser. The second one is a concrete implementation of *BsHTMLParser* class. This class uses the Beautiful Soup 4 external library, and the lxml parser. It allows more comfortable and fast parsing of XML/HTML documents.

The *HTMLParser* class is an abstract class. This class defines these methods:

- *find_elements* - A method for returning a list of elements matched by search criteria. Two parameters specify the criteria, where the first parameter is a tag name and the second is an attribute dictionary. The dictionary item consists of an attribute name, followed by a string that can represent the attribute value or True, which indicates the presence of an attribute inside the element.
- *base* – base URL of the HTML document.
- *charset* – HTML content encoding.
- *export* – transforms a document from internal representation into a string value.

The concrete parser class inherits from the abstract class, and its implementation is in *BsHTMLParser* class. This class requires dependency injection with parser object during its instantiation. The parser object must be an instance from the Beautiful Soup library. This class also implements all abstract methods inherited from the superclass.

The HTML element representation has a similar design to the parser implementation. There is one abstract class *Element* defining an interface. Each type of parser implements its specific element implementation. HTML element classes can vary depending on the type of parser. The abstract class has one parameter (the element reference), and three methods:

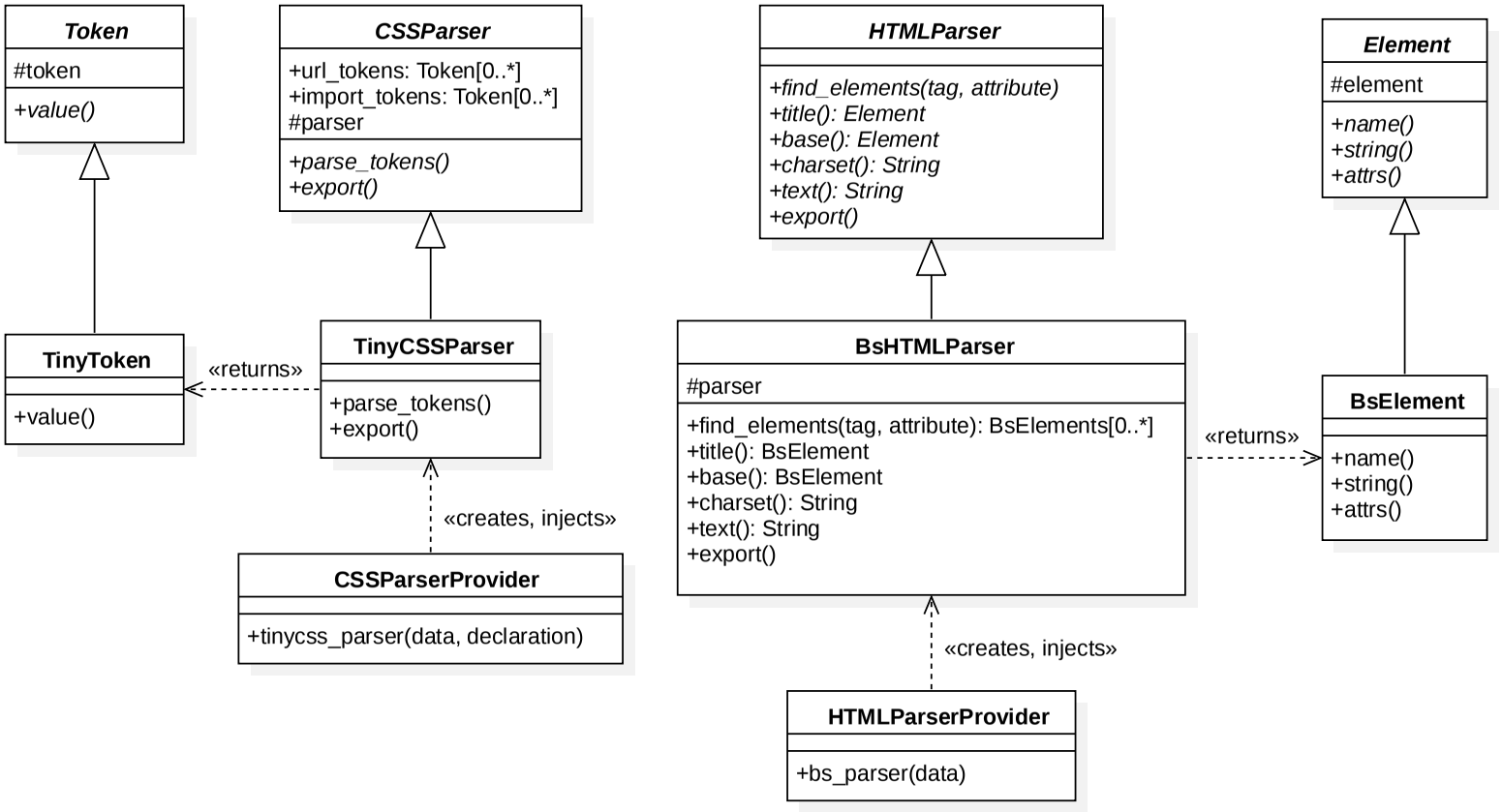



Figure 3.3: Parser package class diagram.

- *name* – the element name.
- *string* – a string value between element tags.
- *atrrs* – a list of elements with their values.

The concrete implementation of the element is in *BsElement* class. This class uses the *BsHTMLParser* class for element representation.

#### CSS parser

CSS parser design is similar to the HTML parser, and there are two main parts: the parser and the token implementation. The parser design has one abstract *CSSParser* class and one concrete implementation. The abstract class defines these methods:

- *parse_tokens* – processes the CSS data, stores URL tokens and import tokens into the list data structure.
- *export* – transforms a document from internal representation into a string value.

The *TinyCSSParser* class inherits from *CSSParser* class. It uses tynycss2 external library. This low-level parser library process CSS data into small fragments. Rules filtrate these fragments. The filtrated fragments or CSS tokens are then accessible by this class.

Similarly, as HTML elements, the CSS tokens are represented by one abstract class and one concrete class. They define and implement the token wrapper for unified use. The token classes have one method, *value*, which returns token value as a string.

#### Providers

A programmer should use the provider classes for parsers instantiation. There are two provider classes: one for HTML parser and another for CSS parser. The names of these classes follow the name convention, where each class name contains the "Provider" substring.

## 3.4 Archive

The archive module implements two archiving modes. Modes reflect web reconstruction types described in Chapter 2. Currently supported archive format is MAFF. This module is designed to archive multiple tabs, which is achieved using general interface implementation. A programmer can implement a custom archiving process using this interface. The archiving process is possible to describe by letters, where each letter represents a set of processes for web content reconstruction. These letters are then stored in an envelope, which is a list data structure. The archiving method processes the envelope, and it results in an archive file.

The archiving method works with HTTP response. The response mostly contains HTML/XML data, where data references on images, videos, etc. are necessary to reconstruct. The entity migration, which contains references manages Migration package. This Python package is part of the archiving module, and detailed description is provided in subsection 3.4.1. Figure 3.4 shows the class diagram of the archive package.

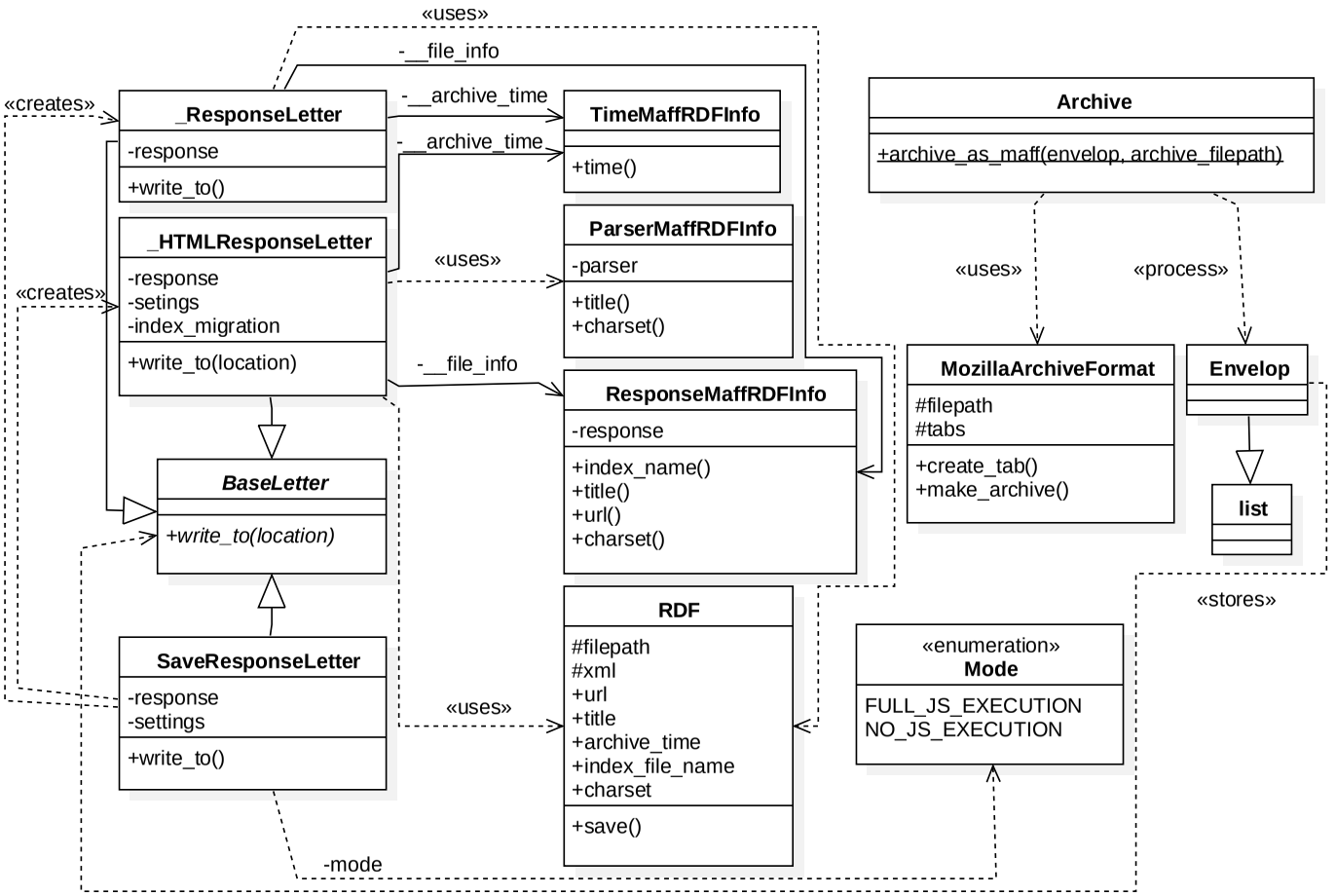



Figure 3.4: Archive package class diagram.

### Meta Information

*RDF* class wraps MAFF meta information. This class encapsulates an XML file. It has several parameters, mostly for setting and fulfilling rows of the RDF file. This class also implements the method *save*, which exports the RDF file to a specified location. The parameters of the *RDF* class are:

- *filepath* – an absolute path into archive directory including file name "index.rdf".
- *xml* – an object generating the XML data.
- *url* – URL address of where the original data are placed.
- *title* – Parameter specifying a title of an HTML document. If the title is not specified, the value can be empty.
- *archive_time* – the time when the archive was created in the format <day-name date time-of-day zone>
- *index_file_name* – the file name with fthe ile extension of an archived file.
- *charset* – specifies an encoding of the index file.

Three classes (*TimeMaffRDFInfo*, *ParserMaffRDFInfo*, and *ResponseMAffRDFInfo*) serve as extractors of RDF data. The first class provides date and time value in a required format. The *ParserMaffRDFInfo* class takes an HTML parser object and extracts the document title and encoding information from it. The *ResponseMAffRDFInfo* class has a *Response* object as an input parameter and gets the index name with a proper file extension, the title of the document, URL address, and data encoding.

### Letters and Envelop

Letters handle reconstruction and archiving. The letters are concrete classes, which inherit the interface from *BaseLetter* abstract class. This class defines one abstract method *write_to*. This method has one argument, which specifies a location, where data should be stored. An envelope is a list data structure, and it keeps all letters.

It is up to the programmer to implement these classes when he or she wants to archive website or other additional information into a MAFF. However, the web page filling process with and without JavaScript preprocessing is already implemented.

The *SaveResponseLetter* class implements web content reconstruction depending on the mode of response preprocessing. This class inherits from BaseLetter class. There are currently two modes of web page reconstruction, and their implementation is in *_ResponseLetter* and *_HTMLResponseLetter* classes. The *SaveResponseLetter* class has three parameters:

- *mode* – contains an enum value from enum Mode class.
- *response* – a *Response* object, which stores data and HTTP status codes.
- *settings* – The parameter has to be an instance of *MigrationSettings class*. This parameter consists of references to resources used for web page reconstruction.

The *_ResponseLetter* class implements archiving of binary data. This class also creates and fulfills the *RDF* class with meta information.

The *_HTMLResponseLetter* class processes HTML and XML documents. This class supports two modes of reconstruction depending on the injected index_migration parameter. Other matching parameters with *SaveResponseClass* class are identical.

#### MAFF archiving

Archiving into MAFF is implemented by the *MozillaArchiveFormat* class. This class wraps the MAFF root directory, creates new subdirectories and ZIP all content to a ZIP file. This class has two parameters: *filepath* and *tabs*, where the first parameter specifies a file name and file location of an archive and the second keeps records of all subdirectories and their content. This class also has two methods:

- *create_tab* – creates a temporary directory and returns its location. The reference to the directory stores tabs list parameter.
- *make_archive* – takes the tabs list class parameter, and all directories (including their content), and compresses everything into a zip file with .maff file extension into a specified location.

### 3.4.1 Migration

This Python package is part of the archive module. It is used for data migration from a given HTML/XML file. The file is parsed, and all elements that require external resources are processed. The external resources can be binary, HTML/XML or CSS data. Some resources can cause recursion, and in this case, the module default processing is into fourth recursion depth.

This module uses all previously described modules such as *HTTP library*, or *Parser* module. It also uses file name generator, which is a directory wrapper, generating unique names inside the specified directory. This file name generator is not part of this package, but its implementation can be found in *filegen.py* in the project root directory.

The migration process also supports two modes, depending on the web page reconstruction mode. If the index file was rendered, the module ignores all included JavaScript and deletes it from the reconstructed content, including all JavaScript events. This approach protects element double rendering in web page content.

The second mode does not pre-render data received from the HTTP GET request. In this mode, all data are reconstructed, including JavaScript. However, the result does not have to reflect the reconstructed content. Whole package diagram is shown in Figures 3.5 and 3.6. The following text describes selected classes and used technologies in more details.

#### Updater

During the content migration, it is necessary to update some of the element attributes. These attributes refer to a location of external resources. After their movement, it is required to update these locations. Because the migration module works with various types of data, to fulfill this requirement the abstract *UpdateEntity* class defines a primary interface and methods for an entity update. This class implements one public method *update_entity* and defines two protected methods. The *update_entity* method has to be

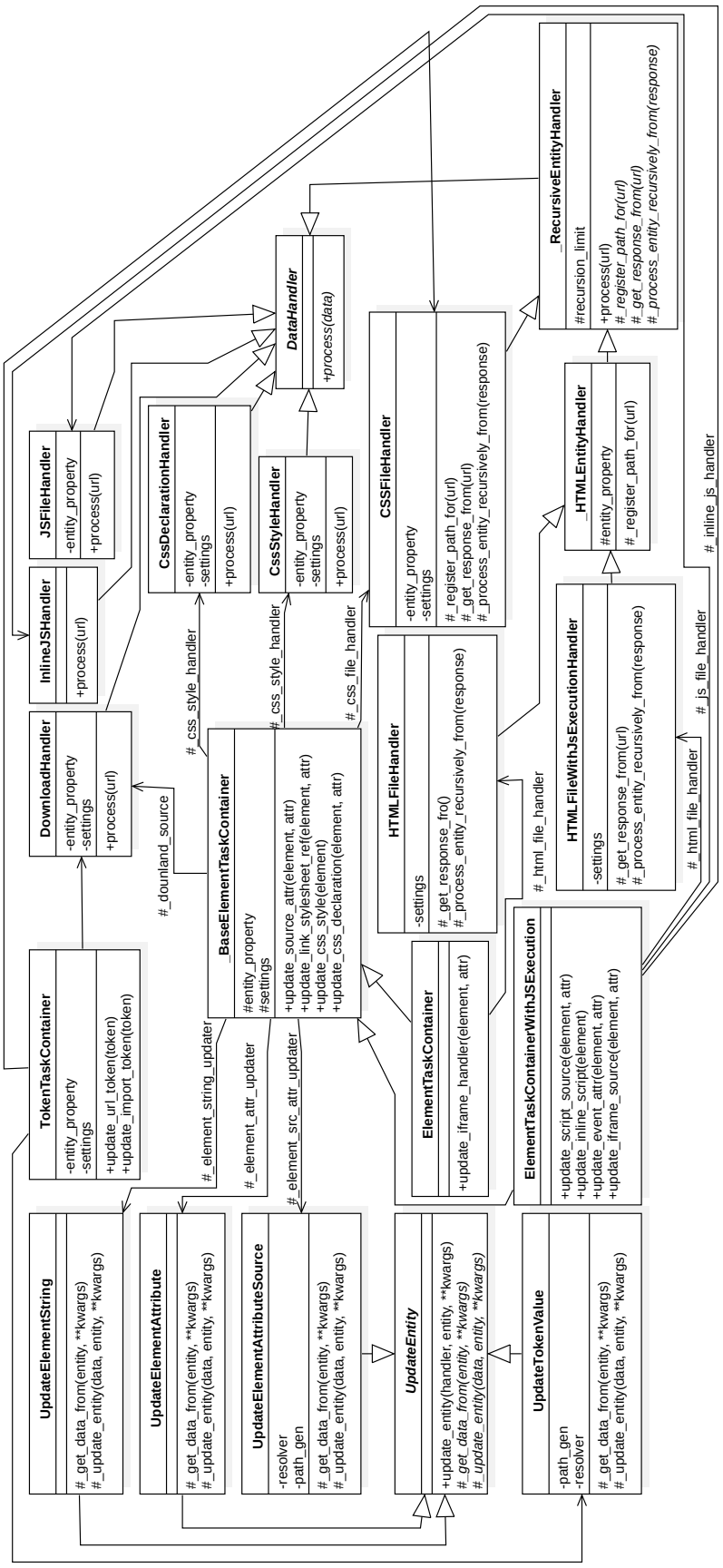



Figure 3.5: Migration module part 1.

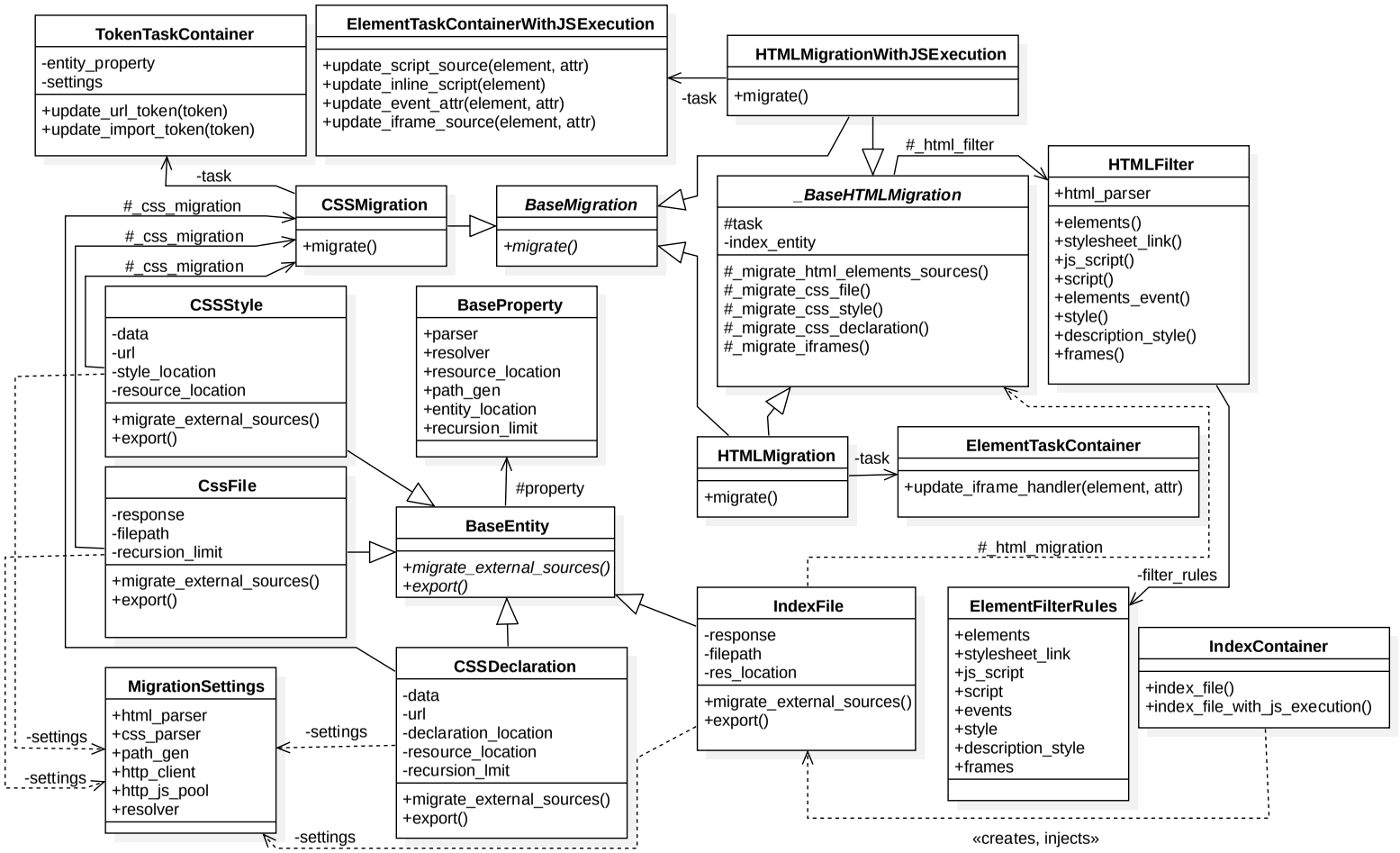



Figure 3.6: Migration module part 2.

injected by a handler object. This object defines how to process a specific type of CSS or HTML elements. All methods of *UpdateEntity* class are:

- *update_entity* – Updates a specific property of an entity. This Entity can be HTML element, or CSS token value. This method uses *_get_data_from* and *_update-entity* methods. These methods are abstract, and they must be implemented by concrete classes. This method has several arguments, where two of them are required. The handler argument specifies handler object, which processes the original entity reference and replaces it with a new one. The second attribute is entity identification. The **kwargs dictionary variable handles optional arguments.

- *_get_data_from* – This abstract method should return required data from an entity as a string value. This method has one required argument, which is the entity object, and optional argument **kwargs, which can be used for entity attribute identification.

- *_update_entity* – An abstract method, which should be implemented by a concrete class and should update the entity property. It has two required arguments. The first is the data argument, which specifies a new value of the entity property. The second is the entity identification. It also has **kwarg argument, which has the same use as in the previous method.

Other classes, which inherit from *UpdateEntity* class, implement previously described abstract methods to match specific requirements of an entity type. For example, UpdateElementAttribute implements the update of an HTML attribute value, or *UpdateTokenValue* updates CSS token value.

### Handler

The handler classes implement data migration from their original location to a specified location. An abstract class *DataHandler* defines an interface for concrete classes. It has one abstract method "process", which migrates data and their new location returned as a string value.

All concrete implementations of *DataHandler* class are used as an argument in *update_entity* methods. This design decision allows implementing various ways how to migrate and satisfy requirements of all elements found during web page archiving. They implement just one process of an abstract method. This method has one argument: the URL address of the resource to migrate. Almost all concrete classes have the *entity_property* parameter (that defines an entity) and the setting parameter (container with all references to HTTP client or parser modules).

To avoid duplicate implementation of code the *_RecursiveEntityHandler* class was implemented. This class handles entities with properties that can cause recursion. Default recursion depth is set to the third level. This class implements the process method and defines three unique protected methods that need different implementation from the concrete classes. Some HTML and CSS elements cause recursion.

### Task Containers

The task containers connect the entity updates and handlers. These classes follow a name convention, where all names of the classes contain the "TaskContainer" substring. There are two types of task containers, one handling CSS tokens, and other handling HTML files.

*TokenTaskContainer* processes CSS tokens. Therefore, it has two methods, one for the URL token, which contains a reference to external resources and another for import tokens, that refer to another CSS file.

The HTML elements processes one base class and two concrete classes. The *_BaseElementTaskContainer* class implements common element updates for both concrete classes. Two concrete classes reflect the mode of accessing the web page. If the content was rendered, then all other recursive resources are rendered as well. Otherwise, the recursive elements are obtained by HTTP request.

### Migration

All migration classes are a generalization of task container classes. Similarly to element migration, they migrate a set of elements like HTML or CSS files. All migration classes contain the "Migration" substring in their names. The migration classes process given data, such as files subset of HTML/CSS documents and transform them into fragments. These fragments processes task containers and all external resources are migrated, and their references are updated.

The migration classes design uses an abstract class to unify the interface for all concrete classes. The *BaseMigration* class is the abstract class defining the abstract method of *migration*. All CSS and HTML migration classes inherit from this class. However, to avoid duplicity during the implementation of HTML element migration, the *_HTMLElementMigration* class implements all common migrations. For this reason, the concrete migration classes for HTML element migration also inherit from this class.

### Entities

Entities represent all HTML parts of the web page. The concrete classes inherit two abstract methods from *BaseEntity* class. This class also creates *BaseProperty* class during instantiation. The *BaseProperty* class is a wrapper of all dependent information used for resources migration, such as a parser, path generator, recursion limit, etc. The following methods need to be implemented by the concrete classes:

- *migrate_external_sources* – this method parses given HTML or CSS data into fragments, which are filtered and processed by migration classes.
- *export* – a method which returns parts or the full document after changes were made on its content. The return value is a string.

It is possible for an entity to identify the index file (or HTML document), the CSS file, a CSS declaration and the body of an HTML style element.

### Filters

The *ElementFilterRules* class contains the list of the entire elements and their attributes. This class implements filter rules for HTML document processing. It also includes JavaScript events of HTML elements. All elements of filter rules are represented as a list of tuples, where each tuple consists of the element name and the attribute dictionary. The attribute dictionary key is the attribute name and the value is either "True" or a string. The "True" value represents the presence of an attribute in the element and a string represents a specific attribute value. Figure 3.7 shows the value structure of the list.

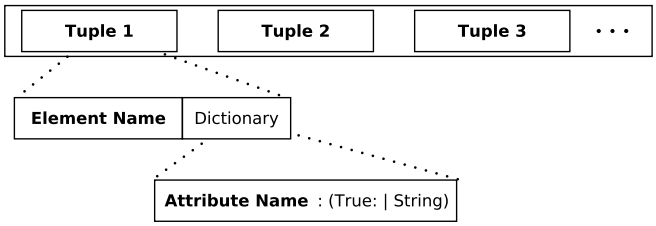



Figure 3.7: List of element filter rules.

Similarly, JavaScript events are represented by a list data structure. Figure 3.8 shows the structure of this list. The tuple representation is different. The first value is "True" value which represents all HTML elements. The second value of the tuple is a dictionary, with the key being the event name and value being "True". The "True" value has the same meaning as in the element filter list.

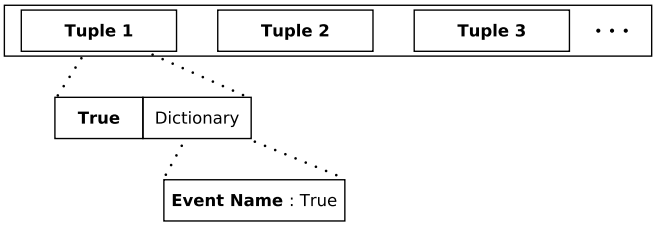



Figure 3.8: List of event filter rules.

The *HtmlFilter* class binds parser and element filter rules together. This class applies the filter rule to data and by implemented methods returns a list of elements fulfilling the filter criteria.

### Provider

This module has just one provider class, which is *IndexContainer*. It has two methods *index_file* and *index_file_with_js_execution*. As the name indicates, both methods create the *IndexFile* object. The difference is in data injection, where the first method creates *IndexFile* without JavaScript pre-processing and the second creates *IndexFile* with JavaScript pre-processing.

## 3.5 Extractor

The last module used for data extraction is the smallest one. It currently supports data extraction by regular expressions. I implemented a generic interface, which allows other developers to implement the extractor upon their needs.

Figure 3.9 shows a class diagram of this Python package. The *RegexExtractor* class inherits abstract extract method from abstract class *BaseExtractor*. This class serves as a generic interface for other extractors. The regular expression class also uses HTML parser to extract textual data from a document. During regular expression extractor object instantiation, three parameters have to be provided:

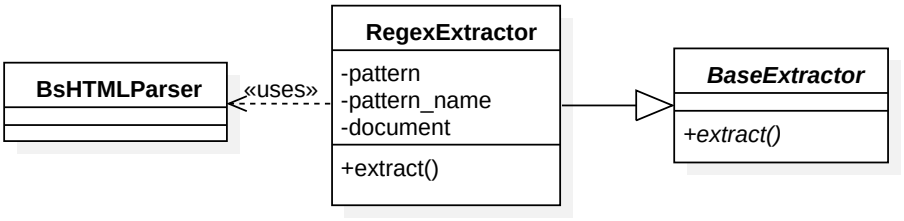



Figure 3.9: Class diagram of the extractor package.

- *pattern* – a regular expression string.
- *pattern_name* – a name identification of the regular expression.
- *document* – a file descriptor.

# Chapter 4

# Applications

This chapter describes two types of applications. The console application and web application. All application types use the Lemmiwings framework as a basic implementation block. The following sections show their basic usage and architecture.

## 4.1 Console

By using the Lemmiwings framework, I developed two console applications. They differ from each other, by the purpose of use. The first application only archive specified web pages and the second one can also extract data from website content and archive them, as well.

Both console applications can pre-render the dynamic web content. Therefore, they need the Selenium standalone drivers. Docker container can provide these drivers, and the user can choose between three types shown in Figure 4.1.

```
$ docker run -d -p 8910:8910 wernight/phantomjs phantomjs --webdriver=8910
# OR
$ docker run -d -p 4444:4444 selenium/standalone-chrome:3.8.1-bohrium
# OR
$ docker run -d -p 4444:4444 selenium/standalone-firefox:3.8.1-bohrium
```

Figure 4.1: Three types of Selenium driver, provided by a docker container.

### 4.1.1 Pharty

This console application can archive web content of specified URL address. It supports two modes of web content reconstruction. The web page can be pre-rendered, or obtained as a response from HTTP GET request. The quality of reconstructed content depends on the type of website. Archiving dynamic content can take at least 4 seconds, but the result is almost the same as the original web content. However, static content is possible to archive without JavaScript preprocessing with the same effect in significantly shorter time.

Table 4.1 describes the script options. All options are possible to combine, and the last two options are required.

| Parameter | Description |
|---|---|
| -h, –help | help message |
| -j | turns on JavaScript pre-rendering during web page archiving |
| -u, –url <address> | URL address |
| -o –output <filepath> | file location, where the archive will be saved |

Table 4.1: The parameters of pharty script

### 4.1.2 Pharty2

The Pharty2 script is designed to be used by the web application MultiFIST described in section 4.2. The script has one argument represented by JSON. The usage example shows Figure 4.2. The script requires URL address followed by output location. For web scraping, it requires an extractor type followed by rule name and rule value.

```
$ pharty2 "{'url': {'https://thepiratebay.org/': '/home/coon/
    ↪ PycharmProjects/multifist/PirateBay'}, 'regex': {'Bitcoin': '(?<=\W)
    ↪ [13][a-km-zA-HJ-NP-Z0-9]{26,33}', 'Litecoin': '(?<=\W)L[a-zA-Z0
    ↪ -9]{26,33}'}}"
```

Figure 4.2: Example of execution of phart2 script

This script also creates two additional tabs during archiving. One is a snapshot of the web page, and the second one is a table with meta information. The script also uses Jinja2 external library to export the information into HTML template. Meta information tab contains IP addresses and other extracted data of the webpage content.

## 4.2 Web application

Multi-functional index scrapping tool is an application providing users administration, task scheduling, and web archiving. It also serves as a search platform over the scraped information. The web application is developed using Django framework **??**.

### 4.2.1 Design and Architecture

The web application was designed according to requirements of legal institutions, which may be the users of this tool. The requirements show use case diagram in Figure 4.3. There are two roles of users, regular user, and administrator. They can create, manage, and edit tasks and rules. The administrator can also manage the user accounts.

The application uses PostgreSQL database. Its database schema shows Figure 4.4. It is necessary to run python manager script, which creates database tables reflecting the ORM of all modules.

The MultiFIST application uses Pharty2 script, described in section 4.1.2. The script is executed as a console application using subprocesses. This web application also uses other external libraries listed below.

- Django [10]
- APSScheduler [11]

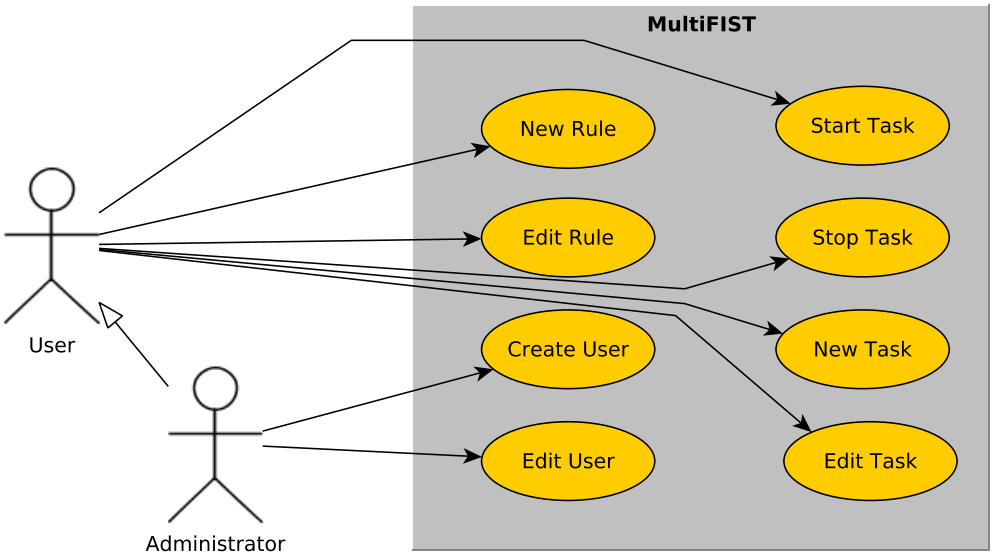



Figure 4.3: Use case diagram of the MultiFIST application

- humanfriendly [21]

### 4.2.2 Scheduler

Task scheduling is handled by APScheduler library [11]. The application supports task creation, task editing, task pausing and resuming. Each task consists of several jobs. An archiving job is executed as a subprocess and is parallelized with other jobs. Each subprocess can handle one web page. A task can process one to five web pages with an unlimited number of scraping rules.

### 4.2.3 UI design

User interface design was meant to be intuitive and straightforward. The prototype application provides several views, where a user can customize the scrapping and archiving process. After signing in to the application, the user is redirected to the task view.

#### Tasks

The task view is the first what user see after authentication. This view consists of the tasks list. As Figure 4.5 shows, a user can also manage the tasks, and create new ones. The task list table includes task name, next task execution, and buttons for task management. The task name also serves as a link to task detail view and web archive list. The task management provides a possibility to pause, restore, edit and delete the job.

The button, on the right corner above task list table, serves as redirection link to form, for new task creation. The form shows Figure 4.6. To create a new task, it is necessary to specify a unique name, interval, scraping rules, and URL addresses. All fields are required. There is an unlimited number of scraping rules to apply on web content.

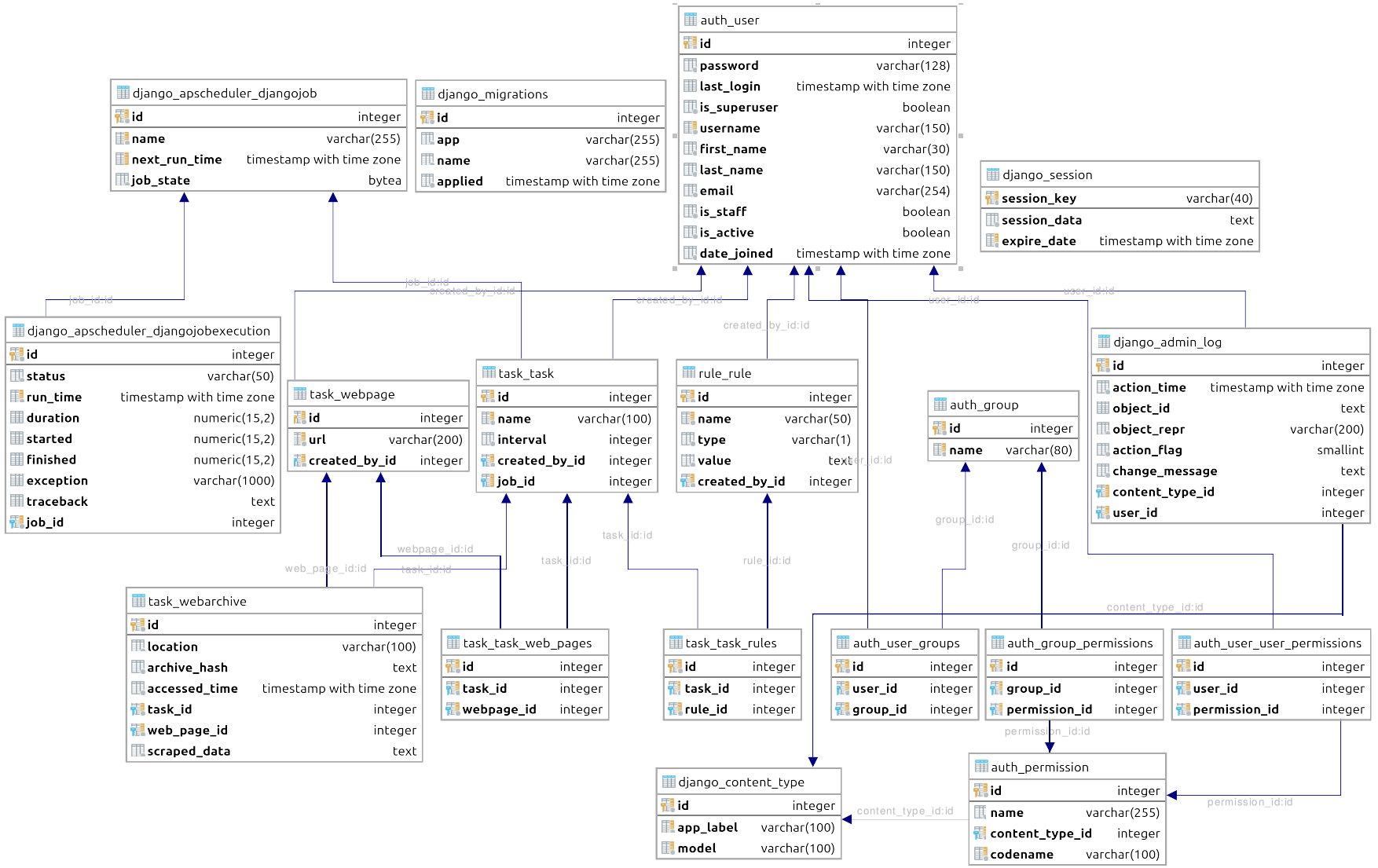



Figure 4.4: The entity–relationship model of the web application

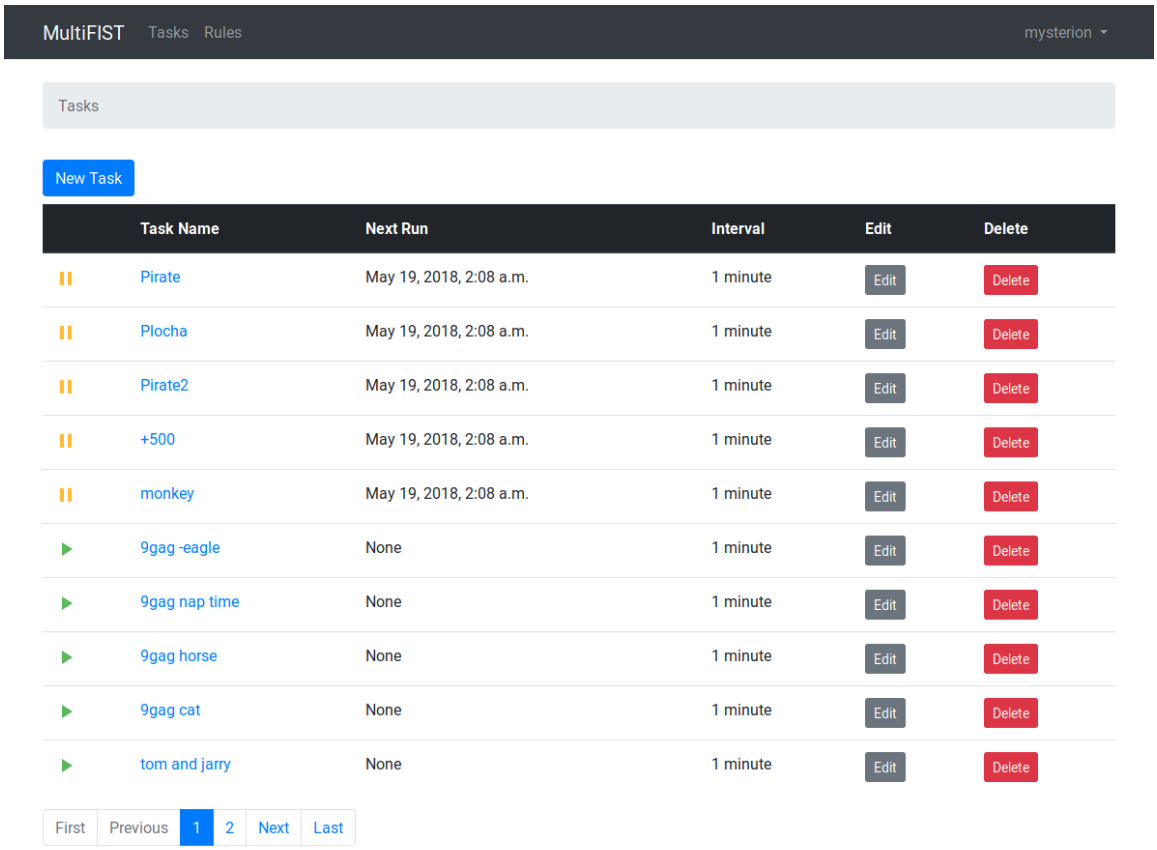



Figure 4.5: Tasks list

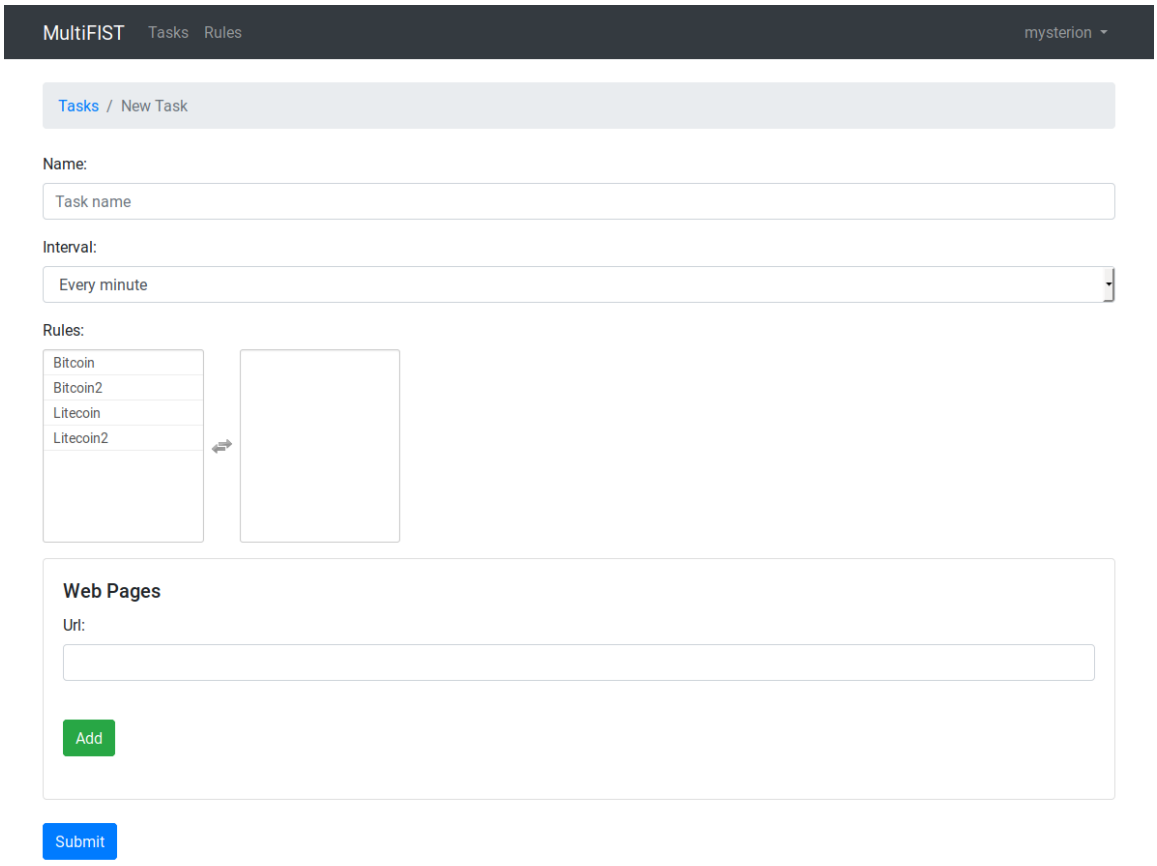



Figure 4.6: New task

### Rules

Rule link in the menu bar redirects users into rules list view. This view shows all specified rules in a table representation. The table shows rule name, its value and managing buttons for editing and deleting. Editable fields of the rule are a name, type, and value. The rules list view shows Figure 4.7.

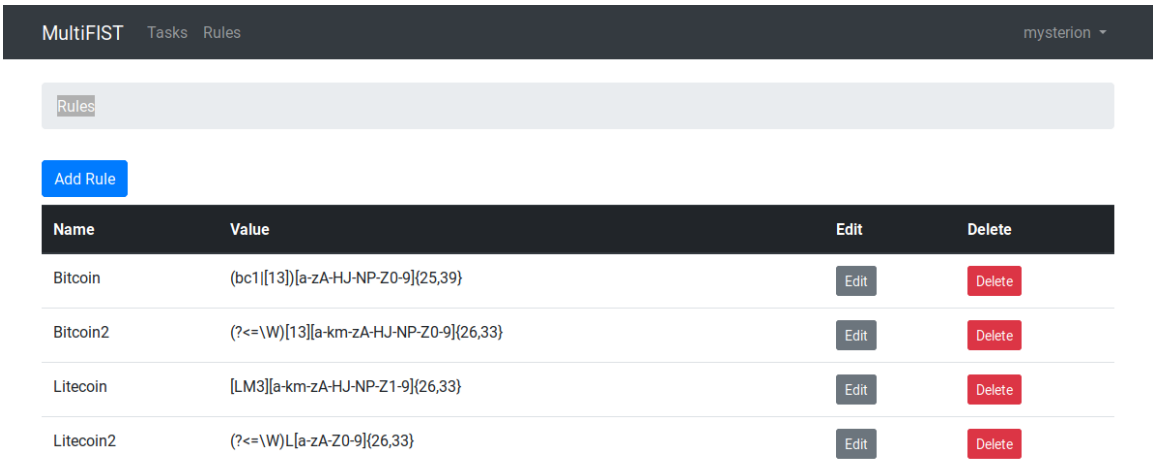



Figure 4.7: Rule list

The link button, in the top left corner of rule list table, redirects users to rule creation view. This view consists of three fields, name, type, and value. All fields are required, and rule name field must be unique. The type field consists of one rule type which specifies rule value to be represented by the regular expression. Figure 4.8 shows the form for creating a new rule.

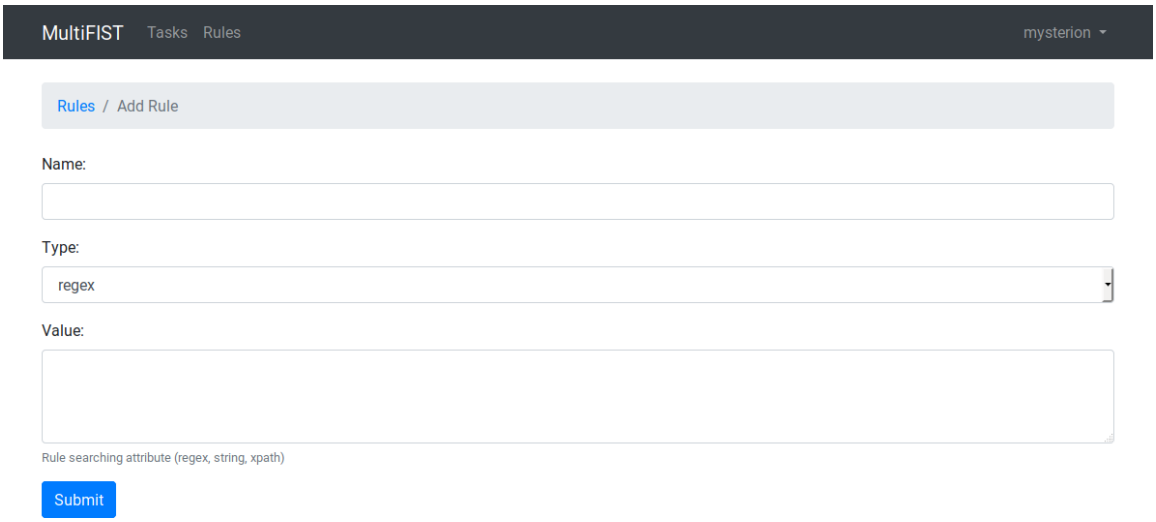



Figure 4.8: New rule

### Web archive

Task detail and web archive list are possible to access through task list view, after clicking on task name. This view shows Figure 4.9, and it includes three panels and one table. The first panel consists of URL addresses, which the task job uses to archive and extract information. The second panel provides rule names used by a task. The last board serves as a searching form, where a user can filtrate the web archives by URL address, time range, and full-text information extracted from a web page. The table shows the list of web archives, including their name, HASH-256 signature, and date time information, when the archive was created. The web archive name also serves as a link to its detail view.

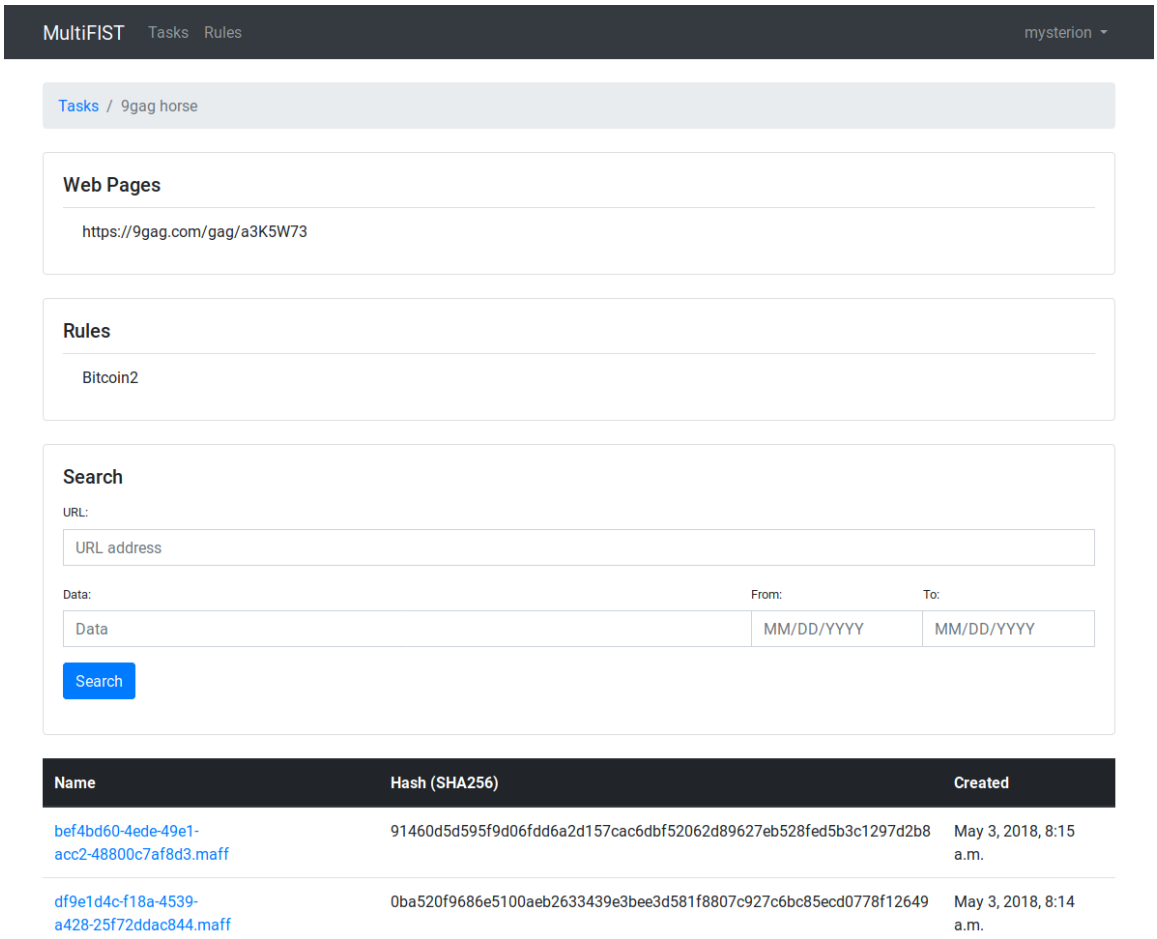



Figure 4.9: Web archive list

The detail view of the web archive shows Figure 4.10. The view contains the table, which provides all scraped information. The particular example shows data scrapped from The Pirate Bay site. Extracted data are Bitcoin and Litecoin addresses. There are also other meta-information like IP addresses and ports, where data was initially placed. The button, in the left bottom corner of the information table, opens the web archive in the browser.

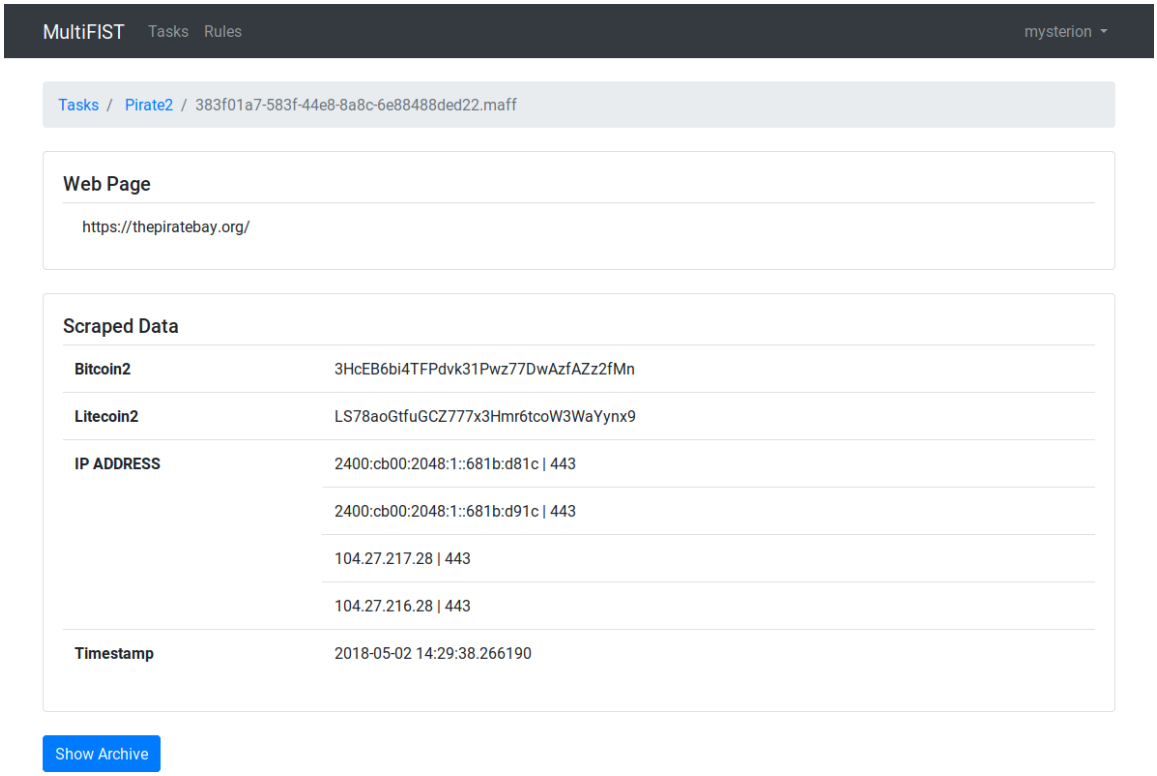



Figure 4.10: Web archive detail

# Chapter 5

# Testing

This chapter aims at results evaluation. The first section demonstrates outcomes using several reconstruction techniques and comparing obtained results on dynamic web content. The second section focuses on web reconstruction applied to content related with cryptocurrencies.

## 5.1 Web reconstruction comparison

The chosen web page of this experiment is accessible on SnapSVG[1] address. The web page shows Figure 5.1. The left menu and the coffee machine are dynamically created elements. The application used for this experiment is Pharty script. It was executed in both reconstruction modes. The results are visually evaluated.

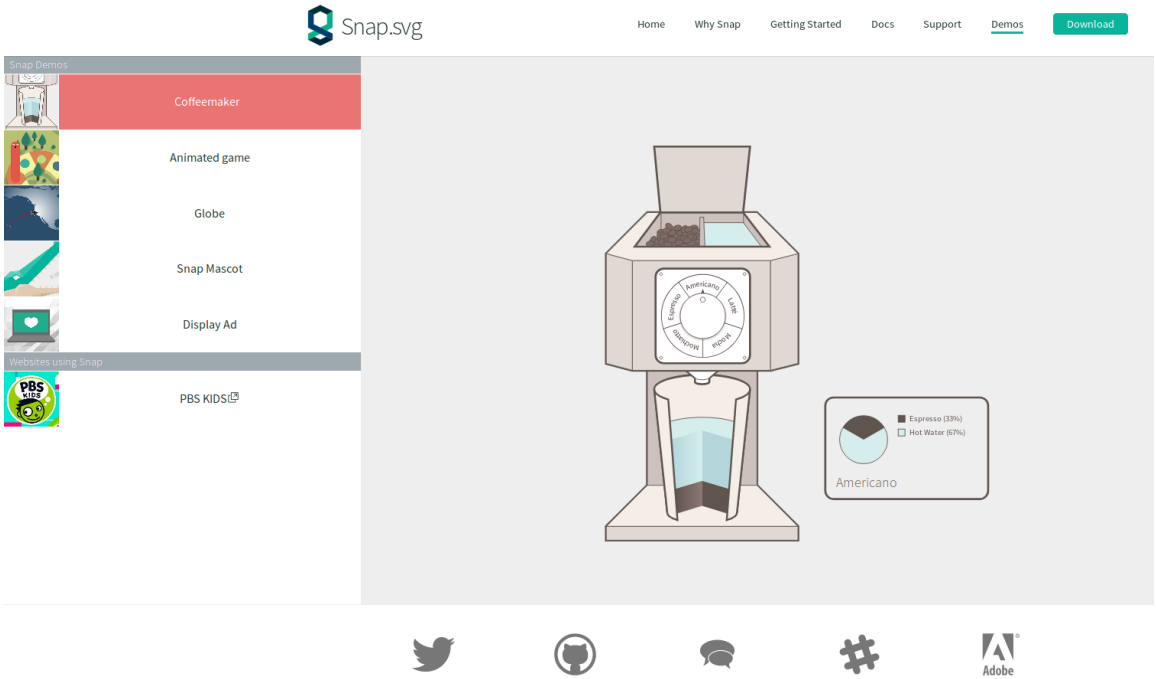



Figure 5.1: Original web page

[1] http://snapsvg.io/demos/address

Figure 5.2 shows reconstructed content without pre-rendering. The final result, compared to the original one, has several defects. Dynamic elements are partially visualized alternatively, missing. The cause of this unfortunate result is missing dependencies, which were not accessible during the reconstruction. JavaScript scripts added the missing references while accessing the archive.

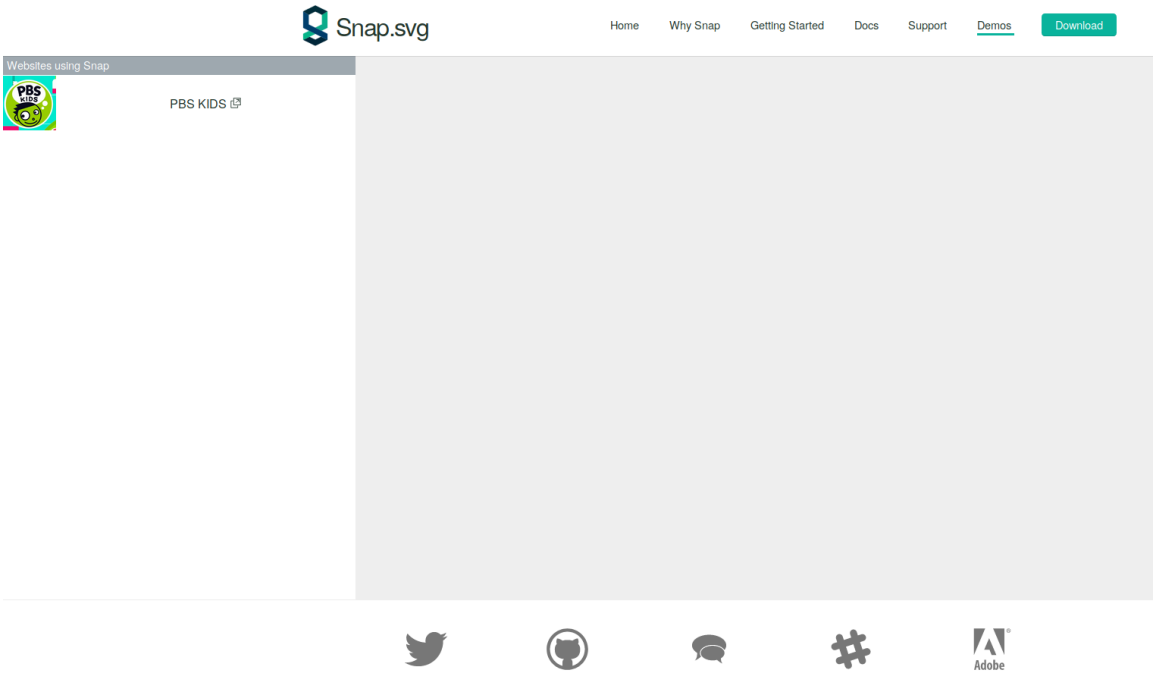



Figure 5.2: The archived web page without pre-rendered content.

The second example shows entirely reconstructed web content in Figure 5.3. The reconstruction result is identical to the original content shown in Figure 5.1. This approach pre-renders web content, and also removes all included JavaScript. If the JavaScript would not be deleted, during archive accessing, some elements may be double rendered as is shown in Figure 5.4. Content, inside the red rectangle mark, represents the duplicate element.

Difference between these two reconstruction methods is noticeable. It is still necessary to distinguish the purpose of use. On static websites, the normal mode works just fine and is faster. For example, on this website, the first script ran approximately 2.5 seconds, while the pre-rendering reconstruction took around 10 seconds.

## 5.2 Cryptocurrency content

Web pages, discussed in this section, were tested on cryptocurrency presence in their content. They were tested using Phart2 script, which allows data extraction. The obtained results were satisfying, compared to the original content. However, the data extractor, using regular expressions, has several defects according to the semantics of the content. In some cases, the extracted information did not match expectations.

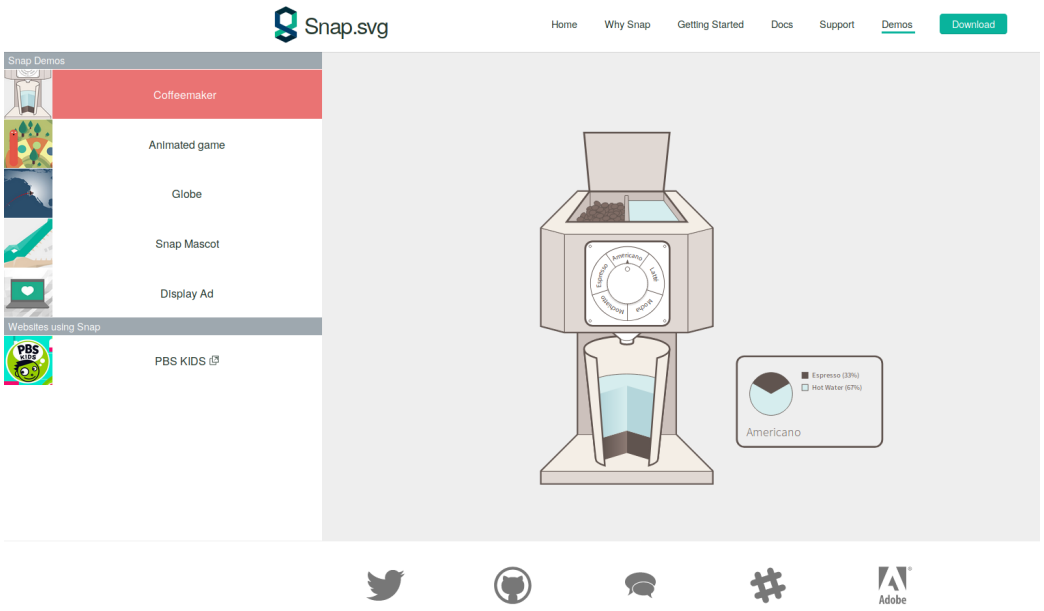



Figure 5.3: The reconstructed web page with pre-rendered content and removed JavaScript.

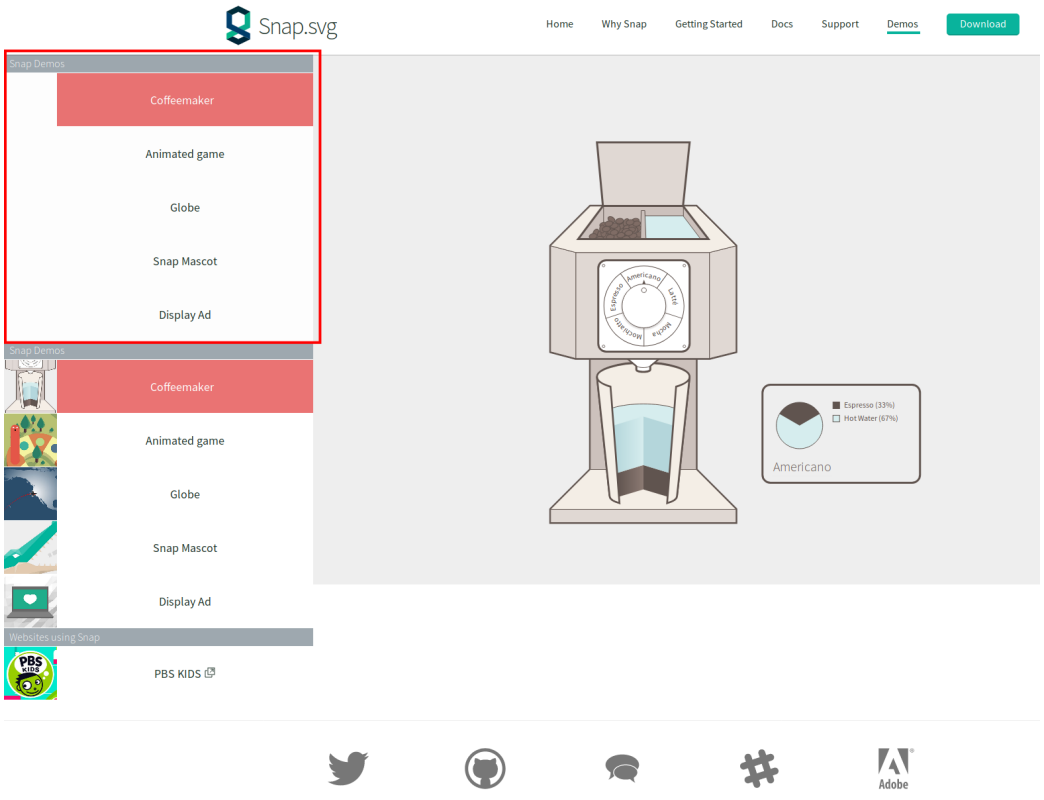



Figure 5.4: The pre-rendered web page without processed JavaScript.

I chose two examples: one is accessible on The Pirate Bay[2] and second on the BlockChain[3] web page. The second web page required a change of the Selenium client interpreter from PhantomJS to Firefox. The reason was the BlockChain website protection against web scraping bots, which use HTTP clients without a user interface.

The first example, shown in Figure 5.5, demonstrates the extraction of the Bitcoin addresses. The website is in The Pirate Bay domain. Color rectangles marks matched data. The red box represents data with a false positive match. I specify the regular expression 5.1 for data extraction. Another possibly adverse occurrence was an impossibility to exclude repetitive information, in this case, the Bitcoin address on website footer, which occurs on each web page of The Pirate Bay domain.

$$\wedge\,[13][a-km-zA-HJ-NP-Z1-9]\{25,34\}\$ \tag{5.1}$$

The second test case shows Figure 5.6. In this case, the reconstruction script was executed several times with different scraping rules. The reconstructed content was all the time the same. The difference comes to the results of extracted information. I was trying to modify scrapping rule to get perfect data match. However, all results were partially correct.

The first used extraction rule was the rule 5.2. The extracted meta information shows Figure 5.6b. Color boxes distinguish all matched Bitcoin addresses. The blue and green box label correctly extracted data. Red and orange box labeled the false positive match. The Bitcoin addresses listed in this figure should correspond to tagged one in Figure 5.6a.

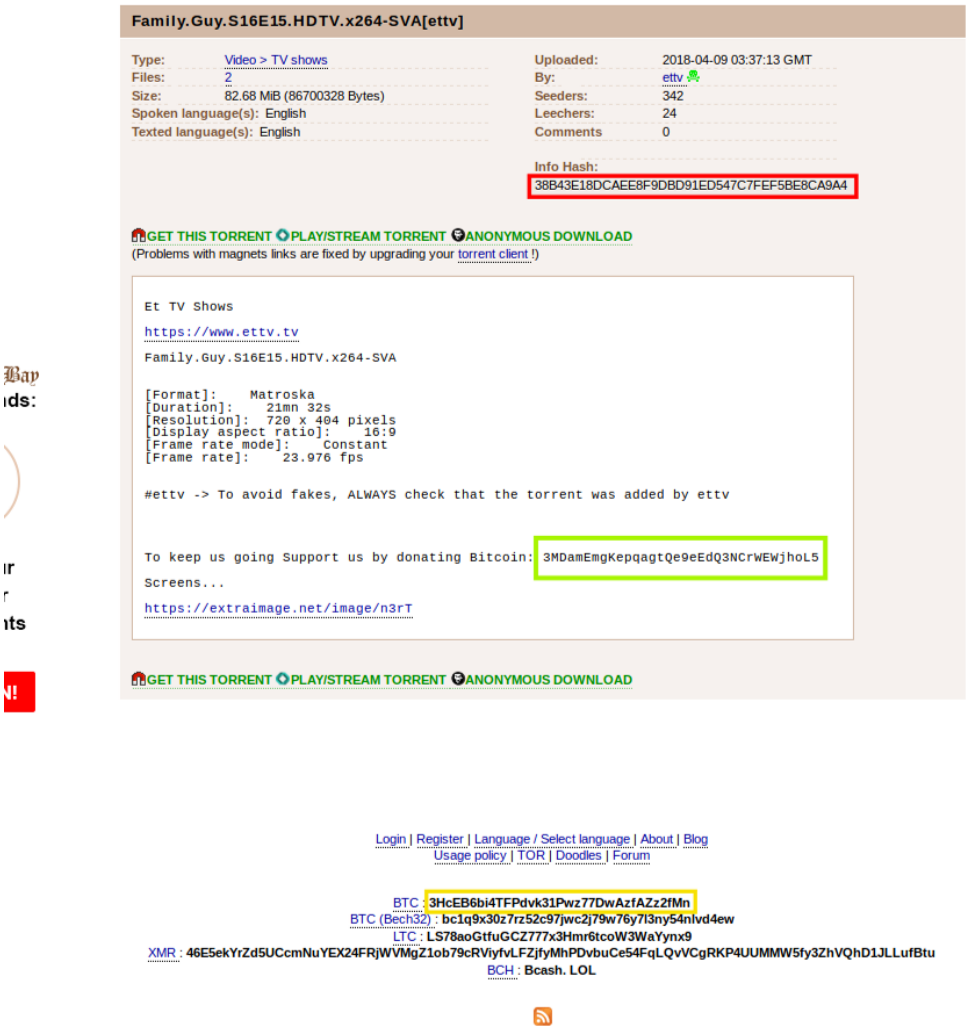
$$(?:bc1|[13])[a-zA-HJ-NP-Z0-9]\{25,39\} \tag{5.2}$$

Modified rule 5.3 differs from the previous one with forward checking. The rule should match all strings starting with "bc1", "1", or "3". The forward checking rule checks if before these strings are whitespaces or not. If there is no other character, then the string satisfying regular expression is matched. This rule did not have false positive matches. However, the result did not extract one valid Bitcoin address. Figure 5.6c shows the meta-information table.

$$\wedge\,(?<=\backslash W)(?:bc1|[13])[a-zA-HJ-NP-Z0-9]\{25,39\} \tag{5.3}$$

The reason for the unfortunate result can be full-text search combine with a regular expression. Data representation can also cause a problem. Their form does not have to hold a traditional structure as written form. They can be part of an element, and when they are merged with other element's data, they can form entire text without spaces. The major problems are user mistakes while composing the regular expression and regular expression limitation. The limitation is intended to content semantics.

[2] https://www.thepiratebay.org/torrent/21016073/Family.Guy.S16E15.HDTV.x264-SVA[ettv]
[3] https://blockchain.info/tx/74be85510ba1f0d6f237f3b3d0bbfe56b37d3258c87e622d56474c58a73c509a





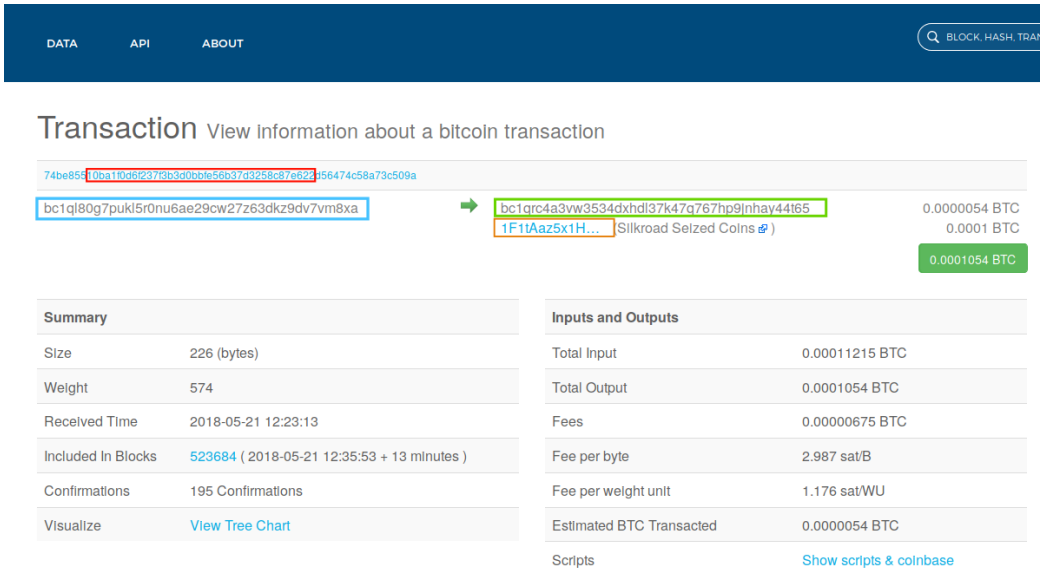


(a) The Pirate Bay website fragment

| Type | Value |
|---|---|
| Bitcoin | 38B43E18DCAEE8F9DBD91ED547C7FEF5BE |
| | 3MDamEmgKepqagtQe9eEdQ3NCrWEWjhoL5 |
| | 3HcEB6bi4TFPdvk31Pwz77DwAzfAZz2fMn |
| IP ADDRESS | 104.27.216.28 \| 443 |
| | 104.27.217.28 \| 443 |
| | 2400:cb00:2048:1::681b:d91c \| 443 |
| | 2400:cb00:2048:1::681b:d81c \| 443 |
| Timestamp | 2018-05-22 17:14:35.784498 |

(b) Extracted information from The Pirate Bay website

Figure 5.5: Example of The Pirate Bay extraction and reconstruction result




(a) Zoomed fragment of the web page

| Type | Value |
|---|---|
| Bitcoin | 10ba1f0d6f237f3b3d0bbfe56b37d3258c87e622 |
| | 10ba1f0d6f237f3b3d0bbfe56b37d3258c87e622 |
| | bc1ql80g7pukl5r0nu6ae29cw27z63dkz9dv7vm8xa |
| | bc1qrc4a3vw3534dxhdl37k47q767hp9jnhay44t65 |
| | 1F1tAaz5x1HUXrCNLbtMDqcw6o5GNn4xqX |
| IP ADDRESS | 104.16.55.3 \| 443 |
| | 104.16.54.3 \| 443 |
| Timestamp | 2018-05-22 19:04:08.981473 |

(b) Meta-information table using rule 5.2

| Type | Value |
|---|---|
| Bitcoin | bc1ql80g7pukl5r0nu6ae29cw27z63dkz9dv7vm8xa |
| IP ADDRESS | 104.16.54.3 \| 443 |
| | 104.16.55.3 \| 443 |
| Timestamp | 2018-05-22 18:39:39.801575 |

(c) Meta-information table using rule 5.3

Figure 5.6: BlockChain's web page example.

# Chapter 6

# Conclusion

This thesis describes a solution for automating web page reconstruction. The result of this work is Lemmiwinks framework two console applications and web application demonstrating framework possible use. The framework supports reconstruction with and without dynamic content pre-rendering.

However, the framework has some limitations when web pages are protected by Captcha, or other advanced tools protecting against the web scraping. These limitations cause problem when the requested content is not received. There are also limitations with data extraction using a regular expression. The flaw is based on data semantics and impossibility to distinguish the meaning of the content, by regular expression.

The advantage of my solution is flexibility during application development. A developer can use the framework to optimize needs of requirements. The framework also provides a sophisticated platform for web archiving and web scraping, which other solutions do not provide. This work was made as part of the solution for project TARZAN[1], which I participate.

## 6.1 Future work

The future work will be focused to optimize Lemmiwinks framework, especially on reconstruction. Selenium client library can also be more efficient. One of the other improvement is the implementation of Captcha resolving and adding a module to allow signing in to private web content. The biggest challenge would be to design and implement the extractor module enabling semantics extraction.

[1] http://www.fit.vutbr.cz/units/UIFS/grants/index.php?id=1063

# Bibliography


[1] Available at https://dexi.io/. online; accessed 8 April 2018.

[2] Available at https://scrapinghub.com/. online; accessed 8 April 2018.

[3] Scrapy. Available at https://scrapy.org/. online; accessed 8 April 2018.

[4] Selenium. Available at https://www.seleniumhq.org/. online; accessed 17 May 2018.

[5] Atkins, T.; Sapin, S.: CSS Syntax Module Level 3. Available at https://drafts.csswg.org/css-syntax-3/. online, accesed 8 April 2018.

[6] Christopher Ottley, P. A.: Features of the MAFF file format. Available at http://maf.mozdev.org/maff-file-format.html. online; accessed 8 April 2018.

[7] Christopher Ottley, P. A.: The MAFF specification. Available at http://maf.mozdev.org/maff-specification.html. online; accessed 8 April 2018.

[8] Christopher Ottley, P. A.: Mozilla Archive Format, with MHT and Faithful Save. Available at https://addons.mozilla.org/en-US/firefox/addon/mozilla-archive-format/. online; accessed 8 April 2018.

[9] contributors, A.: Welcome to AIOHTTP. Available at https://aiohttp.readthedocs.io/en/stable/. online; accessed 8 April 2018.

[10] Foundation, D. S.: The Web framework for perfectionists with deadlines | Django. Available at https://www.djangoproject.com/. online; accessed 22 May 2018.

[11] Grönholm, A.: Advanced Python Scheduler. Available at https://apscheduler.readthedocs.io/en/latest/. online; accessed 8 April 2018.

[12] Grönholm, A.: Asynchronous generators, context managers and other utilities for use with asyncio. Available at https://github.com/agronholm/asyncio_extras. online; accessed 22 May 2018.

[13] Grönholm, A.: pip. Available at https://pip.pypa.io/en/stable/. online; accessed 22 May 2018.

[14] Hupp, A.: Python-magic - A python wrapper for libmagic. Available at https://github.com/ahupp/python-magic. online; accessed 22 May 2018.

[15] Ishida, R.: An Introduction to Multilingual Web Addresses. *updated*. vol. 15. 2008: page 32.

[16] Labs, E.: Dependency injection and inversion of control in Python. Available at http://python-dependency-injector.ets-labs.org/introduction/di_in_python.html. online; accessed 22 May 2018.

[17] Lin, D.: webscrapbook. Available at https://github.com/danny0838/webscrapbook. online; accessed 8 April 2018.

[18] Masinter, L.: RFC 2397: The "data" URL scheme,". *IETF, August*. 1998.

[19] Mozilla: HTML elements reference. Available at https://developer.mozilla.org/en-US/docs/Web/HTML/Element. online; accessed 8 April 2018.

[20] Mozilla; individual contributors: Event reference. Available at https://developer.mozilla.org/en-US/docs/Web/Events. online; accessed 23 May 2018.

[21] Odding, P.: humanfriendly: Human friendly input/output in Python. Available at https://humanfriendly.readthedocs.io/en/latest/. online; accessed 22 May 2018.

[22] Richardson, L.: Beautiful Soup: We called him Tortoise because he taught us. Available at https://www.crummy.com/software/BeautifulSoup/. online; accessed 22 May 2018.

[23] Richter, S.: lxml - Processing XML and HTML with Python. Available at http://lxml.de/. online; accessed 22 May 2018.

[24] Sapin, S.: tinycss2: Low-level CSS parser for Python. Available at http://tinycss2.readthedocs.io/en/latest/. online; accessed 17 May 2018.

[25] Tvrtković, T.: aiofiles: file support for asyncio. Available at https://github.com/Tinche/aiofiles. online; accessed 22 May 2018.

[26] Vesterinen, K.: Python Data Validation for Humans™. Available at http://validators.readthedocs.io/en/latest/. online; accessed 22 May 2018.

# Appendices

# Appendix A

# JavaScript Events

| Event | Description |
|---|---|
| onafterprint | The print dialog is closed |
| onbeforeprint | Document is about to be printed or previewed for printing |
| onbeforeunload | The window, the document and its resources are about to be unloaded |
| onerror | A resource failed to load |
| onhashchange | The fragment identifier of the URL has changed |
| onfocus | An element has received focus (does not bubble) |
| oninput | The value of an <input>, <select>, or <textarea>element is changed |
| oninvalid | A submittable element has been checked and doesn't satisfy its constraints |
| onreset | A form is reset |
| onsearch | A user presses the „ENTER“ key or clicks the „x“ button in an <input>element with type=„search“ |
| onselect | Some text is being selected |
| onsubmit | The submit button is pressed |
| onkeydown | ANY key is pressed |
| onkeypress | ANY key except Shift, Fn, CapsLock is in pressed position. (Fired continously.) |
| onkeyup | ANY key is released |
| onclick | A pointing device button (ANY button; soon to be primary button only) has been pressed and released on an element |
| ondblclick | A pointing device button is clicked twice on an element |
| onmousedown | A pointing device button is pressed on an element |
| onmousemove | A pointing device is moved over an element. (Fired continously as the mouse moves.) |
| onload | A resource and its dependent resources have finished loading |
| onloadedmetadata | The metadata has been loaded |
| onmessage | A message is received through a WebSocket. |

Table A.1: JavaScript events part 1[20]

| Event | Description |
|---|---|
| ononline | the browser has gained access to the network and the value of navigator.onLine switched to true |
| onmouseout | A pointing device is moved off the element that has the listener attached or off one of its children. |
| onmouseover | A pointing device is moved onto the element that has the listener attached or onto one of its children. |
| onmouseup | A pointing device button is released over an element. |
| onmousewheel | a mouse wheel or similar device is operated |
| onwheel | A wheel button of a pointing device is rotated in any direction. |
| ondrag | An element or text selection is being dragged (Fired continuously every 350ms). |
| ondragend | A drag operation is being ended (by releasing a mouse button or hitting the escape key). |
| ondragenter | A dragged element or text selection enters a valid drop target. |
| ondragleave | A dragged element or text selection leaves a valid drop target. |
| ondragover | An element or text selection is being dragged over a valid drop target. (Fired continuously every 350ms.) |
| ondragstart | The user starts dragging an element or text selection. |
| ondrop | An element is dropped on a valid drop target. |
| onscroll | The document view or an element has been scrolled. |
| oncopy | The selection has been copied to the clipboard |
| onpagehide | A session history entry is being traversed from. |
| onpageshow | A session history entry is being traversed to. |
| onpopstate | A session history entry is being navigated to (in certain cases). |
| onresize | The document view has been resized. |
| onstorage | a storage area (localStorage or sessionStorage) has been modified. |
| oncut | The selection has been cut and copied to the clipboard |
| onpaste | The item from the clipboard has been pasted |
| onabort | The loading of a resource has been aborted. |
| oncanplay | The browser can play the media, but estimates that not enough data has been loaded to play the media up to its end without having to stop for further buffering of content. |
| oncanplaythrough | The browser estimates it can play the media up to its end without stopping for content buffering. |
| | |
| ondurationchange | The duration attribute has been updated. |

Table A.2: JavaScript events part 2[20]

| Event | Description |
|---|---|
| onemptied | The media has become empty; for example, this event is sent if the media has already been loaded (or partially loaded), and the load() method is called to reload it. |
| onended | Playback has stopped because the end of the media was reached. |
| onerror | A WebSocket connection has been closed with prejudice (some data couldn't be sent for example). |
| onloadeddata | The first frame of the media has finished loading. |
| onloadstart | Progress has begun. |
| onpause | Playback has begun. |
| onplay | Playback has begun. |
| onvolumechange | The volume has changed. |
| onunload | The document or a dependent resource is being unloaded. |
| onblur | An element has lost focus (does not bubble). |
| onchange | An event is fired for <input>, <select>, and <textarea>elements when a change to the element's value is committed by the user |
| oncontextmenu | The right button of the mouse is clicked (before the context menu is displayed). |
| onplaying | Playback is ready to start after having been paused or delayed due to lack of data. |
| onprogress | In progress. |
| onratechange | The playback rate has changed. |
| onseeked | A seek operation completed. |
| onseeking | A seek operation began. |
| onstalled | The user agent is trying to fetch media data, but data is unexpectedly not forthcoming. |
| onsuspend | Media data loading has been suspended. |
| ontimeupdate | The time indicated by the currentTime attribute has been updated. |
| onwaiting | Playback has stopped because of a temporary lack of data. |
| onshow | A contextmenu event was fired on/bubbled to an element that has a contextmenu |
| ontoggle | The user opens or closes the <details>element. |

Table A.3: JavaScript events part 3[20]